

21天学编程系列

21天学通 C++

第4版

刘蕾 编著

电子工业出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书从初学者的角度较全面地介绍了 C++ 的相关知识, 较系统地介绍了 C++ 语言的基础内容。本书包括 6 篇共 21 章的内容。其中, 第 1 篇是 C++ 数据表达篇, 包括 C++ 入门、变量和数据类型、运算符和表达式以及程序控制结构等; 第 2 篇是 C++ 面向过程设计篇, 包括函数、编译预处理、数组、指针和构造数据类型等内容; 第 3 篇是 C++ 面向对象编程篇, 主要包括类和对象、继承、多态、运算符重载和输入/输出流等内容; 第 4 篇主要介绍了 C++ 高级特性, 内容包括文件、命名空间和引用与内存管理; 第 5 篇的内容主要是 C++ 编程实践, 主要分析了标准模板库 STL、模板与 C++ 标准库和异常处理等; 最后一篇结合学生成绩管理系统开发实例, 讲解如何使用 C++ 进行实际开发。

本书适合没有编程基础的 C++ 语言初学者作为入门教程, 也可作为大中专院校师生和培训班的教材, 对于 C++ 语言开发的爱好者, 本书也有较大的参考价值。

本书附带 DVD 光盘 1 张, 内容包括超大容量教学视频、电子教案 (PPT)、源代码等。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有, 侵权必究。

图书在版编目 (CIP) 数据

21 天学通 C++ / 刘蕾编著. —4 版. —北京: 电子工业出版社, 2016.4

(21 天学编程系列)

ISBN 978-7-121-27879-2

I. ①2… II. ①刘… III. ①C 语言—程序设计 IV.TP312

中国版本图书馆 CIP 数据核字 (2015) 第 304110 号

策划编辑: 牛 勇

责任编辑: 徐津平

印 刷: 北京京科印刷有限公司

装 订: 北京京科印刷有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本: 787×1092 1/16 印张: 26.5 字数: 634 千字

版 次: 2009 年 1 月第 1 版

2016 年 4 月第 4 版

印 次: 2016 年 4 月第 1 次印刷

定 价: 59.80 元 (含 DVD 光盘 1 张)

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zltts@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。

前言

千里之行，始于足下！

——老子

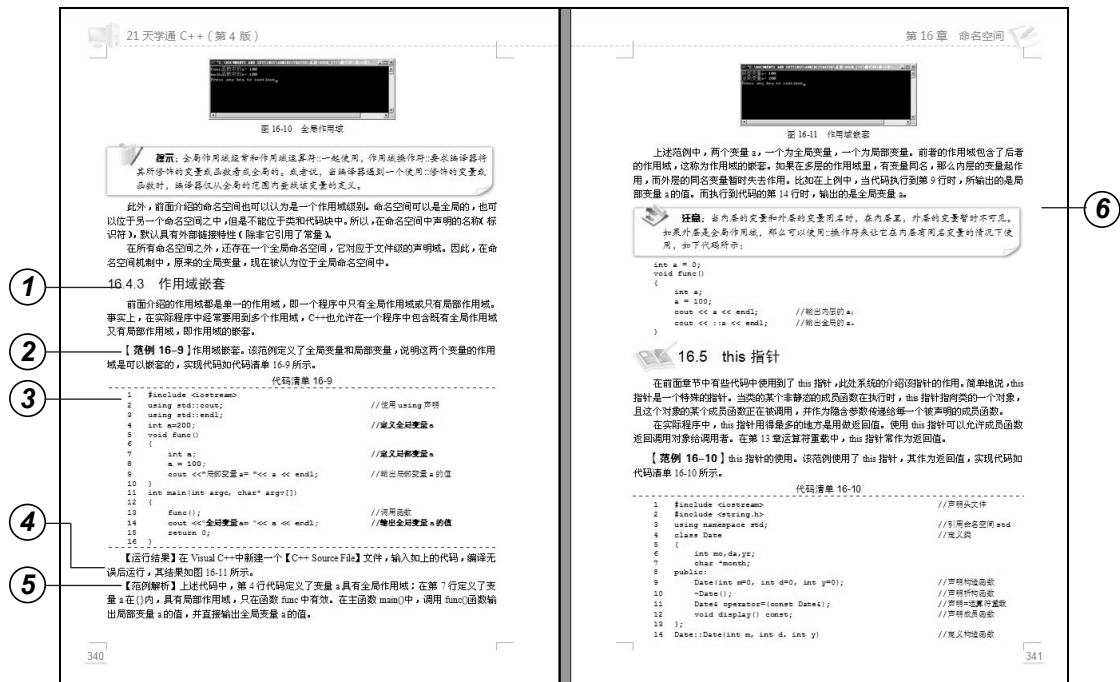
“21 天学编程系列”自 2009 年 1 月上市以来一直受到广大读者的青睐。该系列中的大部分图书从一上市就登上了编程类图书销售排行榜的前列，很多大中专院校也将该系列中的一些图书作为教材使用，目前这些图书已经多次印刷、改版。可以说，“21 天学编程系列”是自 2009 年以来国内原创计算机编程图书最有影响力的品牌之一。

为了使该系列图书能紧跟技术和教学的发展，更加适合读者学习和学校教学，我们结合新技术和读者的建议，对该系列图书进行了改版（即第 4 版）。本书便是该系列中的 C++ 分册。

本书有何特色

1. 细致体贴的讲解

为了让读者更快地上手，本书特别设计了适合初学者的学习方式，用准确的语言总结概念#用直观的图示演示过程#用详细的注释解释代码#用形象的比方帮助记忆。效果如下：



① **知识点介绍** 准确、清晰是其显著特点，一般放在每一节开始位置，让零基础的读者了解相关概念，顺利入门。

② **范例** 书中出现的完整实例，以章节顺序编号，便于检索和循序渐进地学习、实践，放在每节知识点介绍之后。

③ **代码清单** 与范例编号对应，层次清楚、语句简洁、注释丰富，体现了代码优美的原则，有利于读者养成良好的代码编写习惯。对于大段程序，均在每行代码前设定编号便于学习。

④ **运行结果** 对范例给出运行结果和对应图示，帮助读者更直观地理解范例代码。

⑤ **范例解析** 将范例代码中的关键代码行逐一进行解释，有助于读者掌握相关概念和知识。

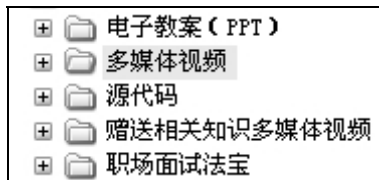
⑥ **贴心的提示** 为了便于读者阅读，全书还穿插着一些技巧、提示等小贴士，体例约定如下：

- **提示：**通常是一些贴心的提醒，让读者加深印象或提供建议，或者解决问题的方法。
- **注意：**提出学习过程中需要特别注意的一些知识点和内容，或者相关信息。
- **警告：**对操作不当或理解偏差将会造成的灾难性后果做警示，以加深读者印象。

经作者多年的培训和授课证明，以上讲解方式是最适合初学者学习的方式，读者按照这种方式会非常轻松、顺利地掌握本书知识。

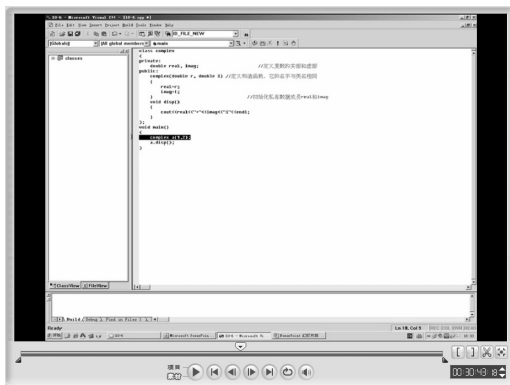
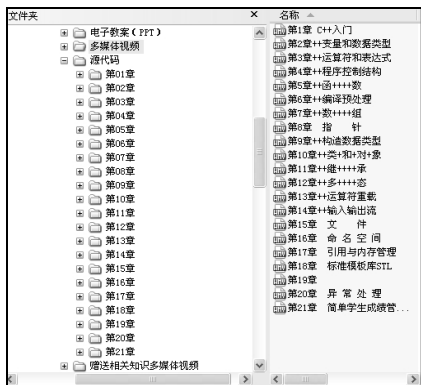
2. 实用超值的 DVD 光盘

为了帮助读者比较直观地学习，本书附带 DVD 光盘，内容包括多媒体视频、电子教案（PPT）和实例源代码等。



● 多媒体视频

本书配有长达 12 小时教学视频，讲解关键知识点界面操作和书中的一些综合练习题。作者亲自配音、演示，手把手教会读者使用。





● 电子教案（PPT）

本书可以作为高校相关课程的教材或课外辅导书，所以作者特别为本书制作了电子教案（PPT），以方便老师教学使用。

● 职场面试法宝

本书附赠“职场面试法宝”，含常见的职场经典面试题及解答。



3. 提供完善的技术支持

本书的技术支持论坛为：<http://www.rzchina.net>，读者可以在上面提问交流。另外，论坛上还有一些教程、视频动画和各种技术文章，可帮助读者提高开发水平。

推荐的学习计划表

本书作者在长期从事相关培训或教学实践过程中，归纳了最适合初学者的学习模式，并参考了多位专家的意见，为读者总结了合理的学习时间分配方式，列表如下：

推荐时间安排		自学目标（框内打钩表示已掌握）	难度指数
第1周	第1天	了解 C++ 的历史及其特点 掌握 C++ 编译环境及第一个 C++ 程序 熟悉 C++ 源程序的基本组成和组成元素	★
	第2天	掌握 C++ 中的常量、变量及其定义 掌握 C++ 中数据类型及其转换 熟练掌握在 C++ 程序中如何声明及使用常量、变量和数据类型	★★
	第3天	掌握 C++ 支持的各种运算符及应用 掌握 C++ 支持的由各种运算符和常量变量构成的表达式、语句及其应用	★★★
	第4天	了解 C++ 的面向过程的结构化设计方法 熟练掌握 C++ 支持的顺序结构、选择结构和循环结构 掌握转向语句的功能及其使用	★★★★
	第5天	掌握 C++ 中函数的声明与定义 熟练掌握函数的参数、原型和返回值，以及在程序中调用函数 了解 C++ 中函数的重载	★★★★★



续表

推荐时间安排		自学目标 (框内打钩表示已掌握)	难度指数
第 1 周	第 6 天	了解预处理命令的功能 <input type="checkbox"/> 掌握宏定义及其使用 <input type="checkbox"/> 掌握文件包含的使用 <input type="checkbox"/> 掌握常用的编译预处理命令 <input type="checkbox"/>	★★★★
	第 7 天	了解数组的概念 <input type="checkbox"/> 熟练掌握一维和多维数组的声明与引用 <input type="checkbox"/> 掌握数组的多种赋值方法 <input type="checkbox"/> 熟悉数组在实际程序中的应用 <input type="checkbox"/>	★★★★★
第 2 周	第 8 天	了解指针的概念 <input type="checkbox"/> 熟练掌握指针的定义和运算 <input type="checkbox"/> 掌握指针与数组、函数和字符串的运算 <input type="checkbox"/> 掌握指向指针的使用 <input type="checkbox"/>	★★★★★★
	第 9 天	掌握结构体、共用体和枚举类型的定义和使用 <input type="checkbox"/> 了解类型重定义符的使用 <input type="checkbox"/> 了解位域的应用 <input type="checkbox"/>	★★★★
	第 10 天	掌握 C++ 中类和对象的概念 <input type="checkbox"/> 掌握 C++ 中类的构造函数、析构函数的定义和应用 <input type="checkbox"/> 掌握友元的概念和相关应用 <input type="checkbox"/>	★★★★★★
	第 11 天	了解 C++ 中继承与派生的概念 <input type="checkbox"/> 掌握 C++ 支持的派生方式 <input type="checkbox"/> 掌握派生类的构造函数和析构函数的定义和使用 <input type="checkbox"/> 掌握多重继承和虚基类的应用 <input type="checkbox"/>	★★★★★★
	第 12 天	理解多态的概念 <input type="checkbox"/> 熟练掌握 C++ 中多态的实现方法 <input type="checkbox"/> 熟练掌握虚函数的定义及其使用 <input type="checkbox"/> 掌握纯虚函数和抽象类 <input type="checkbox"/>	★★★★★★
	第 13 天	理解运算符重载的概念及定义 <input type="checkbox"/> 掌握运算符重载的两种形式及其实现 <input type="checkbox"/> 掌握特殊运算符的重载 <input type="checkbox"/>	★★★★★
	第 14 天	了解 C++ 中引入标准输入/输出流的原因 <input type="checkbox"/> 掌握常用标准输入/输出流对象 <input type="checkbox"/> 掌握输入/输出流成员函数的使用和格式控制 <input type="checkbox"/>	★★★★
	第 15 天	了解文件和流的概念 <input type="checkbox"/> 掌握文件的打开与关闭操作 <input type="checkbox"/> 掌握顺序文件和随机文件的读写及其应用 <input type="checkbox"/>	★★★★
第 3 周	第 16 天	理解命名空间的作用 <input type="checkbox"/> 掌握命名空间的使用方法 <input type="checkbox"/> 掌握类的作用域及 this 指针的应用方法 <input type="checkbox"/>	★★★★



续表



推荐时间安排		自学目标（框内打钩表示已掌握）	难度指数
第 3 周	第 17 天	理解引用的概念 <input type="checkbox"/>	★★★★
		掌握引用在实际程序中的使用和操作及其与指针的区别 <input type="checkbox"/>	
		掌握动态内存分配的方法 <input type="checkbox"/>	
	第 18 天	了解标准模板库 STL 的基本概念及其在 C++程序设计中的作用 <input type="checkbox"/>	★★★★★
		掌握常用的 STL 容器的类别及其相关应用 <input type="checkbox"/>	
		掌握算法和迭代器的使用 <input type="checkbox"/>	
	第 19 天	理解模板的概念 <input type="checkbox"/>	★★★★
		掌握函数模板和类模板的定义和生成 <input type="checkbox"/>	
		理解 C++标准库及字符串库 <input type="checkbox"/>	
	第 20 天	了解错误与异常的概念及其处理基本原则 <input type="checkbox"/>	★★★★
		掌握实际程序中实现异常处理的方法 <input type="checkbox"/>	
		了解异常处理机制 <input type="checkbox"/>	
	第 21 天	了解开发一个应用程序的软件工程生命周期 <input type="checkbox"/>	★★★★
		掌握使用 C++开发具体应用程序的流程 <input type="checkbox"/>	
		掌握使用 Visual C++ 6.0 的控制台程序开发 C++应用程序 <input type="checkbox"/>	

本书适合哪些读者阅读



- 本书非常适合以下几类人员阅读：
- 从未接触过 C++编程，但对 C++有兴趣的自学人员；
 - 各大中专院校的在校学生和相关授课老师；
 - 了解一些 C++，但还需要进一步学习的人员；
 - 在某些需要使用 C++编程的特殊领域的工作人员；
 - 其他编程爱好者。

目 录

第一篇 C++数据表达篇

第1章 C++入门 ( 教学视频: 31 分钟)	1
1.1 C++概述	1
1.1.1 C++的历史沿革	1
1.1.2 C++与面向对象	1
1.1.3 从C到C++	2
1.2 程序设计方法	3
1.2.1 结构化程序设计	3
1.2.2 面向对象程序设计	4
1.2.3 程序设计方法比较	4
1.3 C++开发环境——Visual C++ 6.0	5
1.3.1 工作区	5
1.3.2 编辑区(Editor Area)	6
1.3.3 输出窗口(Output Panel)	6
1.3.4 菜单栏、工具栏、状态栏	7
1.4 第一个C++程序——Hello World	7
1.4.1 创建源程序	7
1.4.2 编译连接	9
1.4.3 调试运行	9
1.5 C++源程序组成元素	10
1.5.1 基本组成	10
1.5.2 基本符号	11
1.5.3 标识符	11
1.5.4 保留字	11
1.5.5 分隔符	12
1.6 小结	12
1.7 习题	12
第2章 变量和数据类型 ( 教学视频: 32 分钟)	16
2.1 常量	16
2.1.1 声明常量	16
2.1.2 常量的应用	17
2.2 变量	19
2.2.1 声明变量	19




2.2.2	变量的命名规则.....	19
2.2.3	变量的作用范围.....	20
2.2.4	变量的应用	21
2.3	基本数据类型.....	22
2.3.1	整型	23
2.3.2	字符型	24
2.3.3	浮点型	26
2.3.4	布尔型	26
2.4	类型转换.....	27
2.4.1	隐式转换	27
2.4.2	显式转换	28
2.5	小结.....	29
2.6	习题.....	29
第 3 章	运算符和表达式 ( 教学视频: 34 分钟)	32
3.1	运算符.....	32
3.1.1	算术运算符	32
3.1.2	赋值运算符	34
3.1.3	关系运算符	35
3.1.4	逻辑运算符	35
3.1.5	条件运算符	36
3.1.6	逗号运算符	37
3.1.7	位运算符	37
3.1.8	sizeof 运算符.....	38
3.1.9	运算符的优先级.....	38
3.2	表达式.....	39
3.2.1	算术表达式	40
3.2.2	关系表达式	40
3.2.3	逻辑表达式	41
3.2.4	条件表达式	42
3.2.5	赋值表达式	43
3.2.6	逗号表达式	43
3.3	语句.....	44
3.3.1	语句中的空格.....	44
3.3.2	空语句	45
3.3.3	声明语句	45
3.3.4	赋值语句	46
3.4	小结.....	46
3.5	习题.....	46
第 4 章	程序控制结构 ( 教学视频: 32 分钟)	50
4.1	顺序结构.....	50
4.1.1	表达式语句	50





4.1.2	输入语句	51
4.1.3	输出语句	51
4.1.4	格式控制符	52
4.1.5	应用示例	55
4.2	选择结构	56
4.2.1	if 语句	56
4.2.2	if...else 语句	57
4.2.3	多重 if...else 语句	58
4.2.4	switch 语句	60
4.2.5	应用示例	62
4.3	循环结构	63
4.3.1	for 语句	63
4.3.2	while 语句	64
4.3.3	do...while 语句	65
4.3.4	多重循环	66
4.3.5	应用示例	67
4.4	转向语句	68
4.5	小结	69
4.6	习题	69



第二篇 C++面向过程设计篇

第5章	函数 ( 教学视频: 36 分钟)	73
5.1	定义函数	73
5.1.1	函数概述	73
5.1.2	定义函数	74
5.1.3	应用示例	75
5.2	函数参数及原型	76
5.2.1	函数的参数及返回值	76
5.2.2	函数原型	77
5.2.3	main()函数	77
5.2.4	带参数的 main()函数	79
5.3	调用函数	80
5.3.1	函数调用格式	80
5.3.2	传值调用	81
5.3.3	引用调用	82
5.3.4	嵌套调用	83
5.3.5	递归调用	83
5.3.6	带默认形参值的函数	85
5.4	变量的作用域	85
5.4.1	局部变量	86
5.4.2	全局变量	87
5.5	函数的作用域	88



5.6	函数重载.....	89
5.6.2	参数类型不同的函数重载.....	90
5.6.3	参数个数上不同的重载函数.....	91
5.7	小结.....	92
5.8	习题.....	92
第 6 章	编译预处理 ( 教学视频: 37 分钟)	95
6.1	预处理命令.....	95
6.2	宏.....	95
6.2.1	宏概述.....	95
6.2.2	不带参数的宏定义.....	96
6.2.3	取消宏.....	97
6.2.4	宏嵌套.....	99
6.2.5	带参数的宏定义.....	99
6.2.6	内联函.....	103
6.2.7	宏与函数的区别.....	104
6.3	文件包含.....	105
6.3.1	#include 命令.....	105
6.3.2	合理使用文件包含.....	106
6.4	条件编译.....	107
6.4.1	#ifdef 形式.....	108
6.4.2	#ifndef 形式.....	109
6.4.3	#if 形式.....	109
6.5	其他命令.....	110
6.5.1	#error 命令.....	110
6.5.2	#line 命令.....	111
6.6	小结.....	111
6.7	习题.....	111
第 7 章	数组 ( 教学视频: 35 分钟)	114
7.1	声明数组.....	114
7.1.1	声明一维数组.....	114
7.1.2	声明多维数组.....	115
7.2	引用数组.....	116
7.2.1	引用一维数组.....	116
7.2.2	引用多维数组.....	117
7.3	数组的赋值.....	118
7.3.1	初始化数组.....	118
7.3.2	通过赋值表达式赋值.....	120
7.3.3	通过输入语句赋值.....	121
7.3.4	通过循环语句赋值.....	121
7.3.5	多维数组的赋值.....	123
7.4	字符串.....	123



7.4.1	传统字符串	124
7.4.2	字符数组	126
7.5	数组与函数	127
7.6	数组应用	129
7.6.1	顺序查找	129
7.6.2	折半查找	130
7.6.3	排序	133
7.7	小结	135
7.8	习题	135
第 8 章	指针 ( 教学视频: 33 分钟)	138
8.1	指针概述	138
8.1.1	指针是什么	138
8.1.2	定义指针	139
8.1.3	指针的初始化	139
8.2	指针的运算	141
8.2.1	取地址与取值运算	141
8.2.2	指针的算术运算	142
8.2.3	指针的关系运算	144
8.2.4	指针的赋值运算	145
8.2.5	void 指针和 const 指针	145
8.3	指针与数组	147
8.3.1	访问数组元素的方法	147
8.3.2	多维数组元素的访问	149
8.3.3	数组指针与指针数组	150
8.4	指针与函数	151
8.4.1	指针作为函数参数	151
8.4.2	指针型函数	153
8.4.3	函数指针	154
8.5	指针与字符串	155
8.6	二级指针	156
8.7	小结	157
8.8	习题	158
第 9 章	构造数据类型 ( 教学视频: 34 分钟)	161
9.1	结构体	161
9.1.1	结构体概述	161
9.1.2	结构体类型说明	162
9.1.3	定义结构体类型变量	163
9.1.4	初始化结构体变量	164
9.1.5	引用结构体成员变量	166
9.1.6	结构体作为函数参数	168
9.2	共用体	169





9.2.1	共用体类型说明.....	169
9.2.2	定义共用体类型变量.....	170
9.2.3	引用共用体成员变量.....	170
9.3	枚举.....	172
9.3.1	定义枚举类型.....	172
9.3.2	定义枚举类型变量.....	174
9.3.3	引用枚举类型变量.....	175
9.4	类型重定义 typedef.....	176
9.5	位域.....	178
9.5.1	定义位域变量.....	178
9.5.2	使用位域.....	179
9.6	小结.....	180
9.7	习题.....	181



第三篇 C++面向对象编程篇

第 10 章	类和对象 ( 教学视频: 35 分钟)	184
10.1	类.....	184
10.1.1	什么是类.....	184
10.1.2	结构到类.....	185
10.1.3	类的声明.....	187
10.1.4	类的访问控制.....	188
10.1.5	类的定义.....	190
10.2	对象.....	192
10.2.1	对象概述.....	192
10.2.2	对象数组.....	193
10.3	构造函数.....	195
10.3.1	构造函数的概念.....	195
10.3.2	构造函数的声明和定义.....	195
10.3.3	构造函数的调用.....	197
10.3.4	不带参数的构造函数.....	197
10.3.5	带有默认参数的构造函数.....	198
10.3.6	构造函数的重载.....	200
10.4	拷贝构造函数.....	201
10.4.1	定义拷贝构造函数.....	201
10.4.2	调用拷贝构造函数.....	202
10.4.3	默认拷贝构造函数.....	204
10.5	析构函数.....	205
10.6	友元.....	206
10.6.1	友元函数.....	206
10.6.2	友元成员.....	208
10.6.3	友元类.....	209
10.7	小结.....	211






10.8 习题	211
第 11 章 继承 ( 教学视频: 38 分钟)	215
11.1 继承与派生	215
11.1.1 继承与派生概述	215
11.1.2 声明派生类	216
11.2 访问控制	216
11.2.1 公有继承	217
11.2.2 私有派生	219
11.2.3 保护继承	220
11.3 派生类的构造函数和析构函数	224
11.3.1 执行顺序和构建原则	224
11.3.2 派生类的构造函数	224
11.3.3 派生类析构函数的构建	225
11.4 多重继承	227
11.4.1 二义性问题	227
11.4.2 声明多重继承	229
11.4.3 多重继承的构造函数和析构函数	231
11.5 虚基类	233
11.5.1 虚基类的引入	233
11.5.2 定义虚基类	234
11.5.3 虚基类的构造函数和初始化	236
11.6 小结	237
11.7 习题	237
第 12 章 多态 ( 教学视频: 34 分钟)	242
12.1 多态	242
12.1.1 什么是多态	242
12.1.2 多态的作用	243
12.1.3 多态的引入	243
12.2 函数重载	245
12.3 虚函数	246
12.3.1 虚函数的引入	246
12.3.2 定义虚函数	248
12.3.3 使用虚函数	249
12.3.4 多重继承和虚函数	250
12.3.5 虚函数的注意事项	252
12.4 抽象类	252
12.4.1 纯虚函数	252
12.4.2 抽象类	254
12.5 小结	255
12.6 习题	256



第 13 章 运算符重载 ( 教学视频: 31 分钟)	259
13.1 运算符重载简介	259
13.1.1 运算符重载的定义	259
13.1.2 运算符重载的特点	260
13.1.3 运算符重载的规则	261
13.2 运算符重载的形式	262
13.2.1 重载为类的成员函数	262
13.2.2 双目运算符重载为成员函数	262
13.2.3 单目运算符重载为成员函数	263
13.2.4 运算符重载为类的友元函数	265
13.2.5 双目运算符重载为友元函数	265
13.2.6 单目运算符重载为友元函数	266
13.2.7 成员运算符函数与友元运算符函数的比较	268
13.3 特殊运算符的重载	268
13.3.1 “++” 和 “--” 的重载	268
13.3.2 赋值运算符 “=” 的重载	270
13.3.3 下标运算符 “[]” 的重载	272
13.4 类类型转换	273
13.5 小结	275
13.6 习题	275
第 14 章 输入/输出流 ( 教学视频: 34 分钟)	279
14.1 输入/输出流的引入	279
14.1.1 printf 与 scanf 的缺陷	279
14.1.2 输入/输出流简介	280
14.1.3 输入/输出流类层次	281
14.2 标准输入/输出流	282
14.2.1 标准输出流对象	282
14.2.2 标准输入流对象	283
14.2.3 标准错误输出流对象	284
14.3 输入/输出流成员函数	285
14.3.1 get() 函数: 输出字符串	285
14.3.2 getline() 函数: 获取字符串	287
14.4 输入/输出的格式控制	287
14.4.1 用 ios 类的成员函数进行格式控制	287
14.4.2 使用格式控制符进行格式控制	290
14.5 用户自定义数据类型的输入/输出	292
14.5.1 重载输出运算符 “<<”	292
14.5.2 重载输入运算符 “>>”	293
14.6 小结	295
14.7 习题	295





第四篇 C++高级特性篇

第 15 章 文件 ( 教学视频: 28 分钟)	298
15.1 文件和流	298
15.1.1 文件概述	298
15.1.2 文件流类	299
15.2 文件的打开与关闭	300
15.2.1 打开文件	300
15.2.2 关闭文件	302
15.3 文件的顺序读写	303
15.3.1 读写文本文件	303
15.3.2 文本文件应用示例	304
15.3.3 二进制文件概述	306
15.3.4 读写二进制文件	306
15.4 文件的随机读写	309
15.5 小结	311
15.6 习题	311
第 16 章 命名空间 ( 教学视频: 34 分钟)	314
16.1 命名空间	314
16.1.1 什么是命名空间	314
16.1.2 定义命名空间	315
16.2 使用命名空间	317
16.2.1 使用作用域运算符引用成员	317
16.2.2 使用 using 指令	318
16.2.3 使用 using 声明	319
16.2.4 using 指令与 using 声明的比较	320
16.3 类的作用域	320
16.3.1 静态数据成员	320
16.3.2 静态成员函数	322
16.4 作用域	323
16.4.1 局部作用域	323
16.4.2 全局作用域	325
16.4.3 作用域嵌套	326
16.5 this 指针	327
16.6 小结	329
16.7 习题	329
第 17 章 引用与内存管理 ( 教学视频: 31 分钟)	332
17.1 引用	332
17.1.1 引用概述	332
17.1.2 引用的使用	333
17.2 引用的操作	334



17.2.1	引用作为函数参数.....	334
17.2.2	引用作为返回值.....	335
17.3	动态内存分配.....	336
17.3.1	申请动态内存.....	336
17.3.2	释放空间	337
17.3.3	malloc 和 free 库函数.....	339
17.4	const 引用	340
17.5	指针与引用的区别	342
17.6	小结.....	343
17.7	习题.....	343


第五篇 C++编程实践篇

第 18 章	标准模板库 STL ( 教学视频: 30 分钟)	346
18.1	标准模板库	346
18.1.1	STL 概述	346
18.1.2	STL 的引入	346
18.1.3	STL 的组成	347
18.2	算法	348
18.3	容器	350
18.3.1	什么是容器	350
18.3.2	向量	350
18.3.3	列表	352
18.3.4	集合	354
18.3.5	双端队列	355
18.3.6	栈	356
18.3.7	映射和多重映射.....	357
18.4	迭代器	358
18.5	小结.....	360
18.6	习题.....	360
第 19 章	模板与 C++标准库 ( 教学视频: 33 分钟)	363
19.1	模板概述.....	363
19.1.1	模板简介	363
19.1.2	模板的引入	364
19.2	函数模板.....	365
19.2.1	定义函数模板.....	365
19.2.2	生成模板函数.....	366
19.2.3	函数模板的异常处理.....	367
19.3	类模板.....	368
19.3.1	定义类模板	368
19.3.2	模板类	369
19.4	C++标准库概述.....	371



19.5	字符串库	372
19.5.1	读写字符串	372
19.5.2	字符串赋值	372
19.5.3	字符串比较	373
19.5.4	字符串长度和空字符串	374
19.6	小结	376
19.7	习题	376
第 20 章	异常处理 ( 教学视频: 31 分钟)	379
20.1	错误与异常	379
20.1.1	什么是异常	379
20.1.2	异常处理的基本思想	379
20.2	异常处理的实现	381
20.2.1	使用 try/catch 捕获异常	381
20.2.2	使用 throw 抛出异常	382
20.2.3	应用示例	383
20.3	类和结构的异常处理	384
20.3.1	异常处理中的构造和析构	385
20.3.2	处理结构类型的异常	386
20.4	异常处理机制	387
20.5	小结	389
20.6	习题	389

第六篇 实例篇

第 21 章	简单学生成绩管理系统开发实例 ( 教学视频: 31 分钟)	392
21.1	需求分析	392
21.2	总体设计	392
21.3	功能模块实现	393
21.3.1	成绩录入模块	394
21.3.2	成绩统计模块	394
21.3.3	成绩排名模块	395
21.3.4	成绩查询模块	397
21.3.5	输出模块	398
21.4	系统集成	398
21.5	系统实现	400
21.5.1	结构和变量定义部分	401
21.5.2	功能函数定义部分	402
21.5.3	主函数部分	403
21.6	小结	405

第一篇 C++数据表达篇

第 1 章 C++入门

C++语言是在 C 语言的基础上增加了面向对象程序设计的要素而发展起来的,本章将介绍 C++的特点,以及其与 C 语言的区别。此外,将着重介绍 C++的编译环境及使用该环境进行第一个 C++程序的设计。在该设计基础上,重点介绍 C++源程序的基本组成和基本元素。以下是对读者在学习本章内容时所提出的几个基本要求,也是本章希望能够达到的目的,让读者在学习本章内容时可以作为学习参照。

- 了解 C++的历史及其特点。
- 掌握 C++编译环境及第一个 C++程序。
- 熟悉 C++源程序的基本组成和组成元素。

1.1 C++概述

C++语言是一种应用较广的面向对象的程序设计语言,其除了继承了 C 语言全部的优点和功能外,还支持面向对象程序设计。C++现在已成为介绍面向对象程序设计的首选语言,也是当前一种十分流行和实用的程序设计语言。

1.1.1 C++的历史沿革

读者可能了解到,C++语言起源于C语言。1980年,美国贝尔实验室的 Bjarne Stroustrup 博士及其同事在 C 语言的基础上,从 Simula 67 中引入面向对象的特征,开发出一种过程性与对象性相结合的程序设计语言。最初称为“带类的 C”,至 1983 年取名为 C++。

后来,Stroustrup 和他的同事们又为 C++引进了运算符重载、引用、虚函数等许多特性,使之更加精练,于 1989 年后推出了 AT&T C++ 2.0 版。随后美国国家标准化协会 ANSI(American National Standard Institute)和国际标准化组织 ISO(International Standards Organization)一起进行了标准化工作,并于 1998 年正式发布了 C++语言的国际标准 ISO/IEC: 98—14882。各软件商推出的 C++编译器都支持该标准,并有一定程度的拓展。

此后,C++经过了许多次改进、完善,发展成为现在的 C++。目前的 C++具有两方面的特点:其一,C++是 C 语言的超集,因此其能与 C 语言兼容;其二,C++支持面向对象的程序设计,使其被称为一种真正意义上的面向对象程序设计语言。

C++支持面向对象的程序设计方法,特别适合于中型和大型的软件开发项目。从开发时间、费用到软件的重用性、可扩充性、可维护性和可靠性等方面,C++均具有很大的优越性。



提示 C++可以认为是 C 语言的一个超集,这就使得许多 C 代码不经修改就可被 C++的编译器编译通过。

1.1.2 C++与面向对象

由于 C++是一种面向对象的程序设计语言,因此具有面向对象程序设计有别于过程化设计

的特点。面向对象程序设计是一种程序设计方法,其模仿了人们建立现实世界模型的方法。在面向对象程序设计中,现实世界中客观存在的事物都被称为对象,而具有相同特征的一类对象则可归纳为类。例如,张三是一个对象,而人则是一个类。面向对象程序设计的基础是对象和类。

C++中,对象是构成信息系统的基本单位,类(class)是对一组性质相同对象的描述。简单地说,类是用户定义的一种新的数据类型,是C++程序设计的核心。由于C++是一种面向对象语言,因此,面向对象程序设计的主要特征也是C++的主要特点,具体如下。

- 封装性:所谓封装就是将一组数据和与这组数据有关的操作集合组装在一起,形成一个能动的实体,也就是对象。C++中通过建立类这个数据类型来支持封装性。
- 继承性:继承是指一个类具有另一个类的属性和行为。这个类既具有另一个类的全部特征,又具有自身的独有特征。C++中将其称为派生类(或子类),而将其所继承的类称为基类(父类)。
- 多态性:多态是指不同的对象调用相同名称的函数,并可导致完全不同的行为。C++中的多态性通过使用函数重载、模板和虚函数等概念来实现。

近几年来,C++得到过许多扩展,使其具有更多独有的特点。C++模板是近几年来对此语言的一种扩展,模板是根据类型的参数来产生函数和类的机制,有时也称模板为“参数化的类型”。使用模板,可以设计一个对许多类型数据进行操作的类,而不需要为每个类型的数据建立一个单独的类。标准模板库(Standard Template Library, STL)和微软的活动模板库(Active Template Library, ATL)都基于C++语言扩展,这些在后续的章节中都将逐一介绍。

此外,C++标准可分为两部分:C++语言本身和C++标准库。C++标准库对于Visual C++是相当新的,C++标准库实现容器和算法的部分就是标准模板库STL。

标准模板库STL是数据结构和算法的一个框架,数据结构包括矢量、列表和映射等,算法包括这些数据结构的查找、复制和排序等。1994年7月,ANSI/ISO C++标准委员会投票决定接受STL为C++标准库的一部分,STL的产生是为了满足通用性的设计目标,而不是为了提高性能。

1.1.3 从C到C++

前面提到过,C++语言是对C语言的扩展,是C语言的超集。C++语言增强了C语言的能力,使得程序员能够提高编写程序的质量,并易于程序代码的复用。C++语言的ISO标准已在1997年11月被一致通过,于1998年8月被正式批准。

事实上,“C++”这个名字是由Rick Maseitti提出,到1983年夏被确定的。C++的创作灵感来源于当时计算机语言多方面的成果,特别是BCPL语言(Basic Combined Programming Language,它也是C语言的来源之一)和Simula 67语言(以面向对象为核心的语言),同时还借鉴了Algol 68。就如同它的名字表达的那样,C++语言是C语言的一个超集,它是一门混合型的语言,既支持传统的结构化程序设计,又支持面向对象的程序设计,这是C++语言成功流行的一个重要原因。



注意 C语言是结构化和模块化的语言,它是面向过程的。C++保留了C语言原有的所有优点,增加了面向对象的机制。

简单来说,C++与C完全兼容。C++既可用于结构化程序设计,又可用于面向对象的程序设计。C++对C的增强和扩展,主要表现在两个方面:

- 在原来面向过程的机制基础上,对C语言的功能做了不少扩充。
- 增加了面向对象的机制。

具体来说,C++与C相比,其优点在于:

- 与C语言兼容,既支持面向对象的程序设计,也支持结构化的程序设计。同时,熟悉C语言的程序员,能够迅速掌握C++语言。



- 修补了 C 语言中的一些漏洞,提供更好的类型检查和编译时的分析。使得程序员在 C++ 环境下继续写 C 代码,也能得到直接的好处。
- 生成目标程序质量高,程序执行效率高。一般来说,用面向对象的 C++编写的程序执行速度与 C 语言程序不相上下。
- 提供了异常处理机制,简化了程序的出错处理。利用 `throw`、`try` 和 `catch` 关键字,使出错处理程序不必与正常的代码紧密结合,提高了程序的可靠性和可读性。
- 函数可以重载。重载允许相同的函数名具有不同参数表,系统根据参数的个数和类型匹配相应的函数。
- 提供了模板机制。模板包括类模板和函数模板两种,它们将数据类型作为参数。对于具体数据类型,编译器自动生成模板类或模板函数,它提供了源代码复用的一种手段。

1.2 程序设计方法

程序设计 (Programming) 是指设计、编制、调试程序的方法和过程。按照结构性质,有结构化程序设计与非结构化程序设计之分。前者是指具有结构性的程序设计方法与过程。它具有由基本结构构成复杂结构的层次性,后者反之。按照用户的要求,有过程式程序设计与非过程式程序设计之分。前者是指使用过程式程序设计语言的程序设计,后者指非过程式程序设计语言的程序设计。下面简单介绍两种程序设计方法:结构化程序设计和面向对象程序设计。

1.2.1 结构化程序设计

在程序设计方法的历史上,最早提出的方法是结构化程序设计,其核心是将程序模块化。早在 1968 年,Dijskstra 就提到了结构化程序设计,之后引起了业内关于 `GOTO` 语句的讨论。经过不断地改进,结构化程序设计日臻完善。

提示 结构化程序设计方法主要使用顺序、选择、循环三种基本结构,形成具有复杂层次的结构化程序,此外,该方法要求严格控制 `GOTO` 语句的使用。

简单地说,结构化程序设计方法的主要特征如下:

- 采用“自顶而下,逐步求精”的设计思想。自顶而下是指从问题的总体目标开始,逐步求精是指层层分解和细化。
- “独立功能,单出、入口”的模块仅用三种(顺序、分支、循环)基本控制结构的编码原则。“独立功能,单出、入口”的模块结构减少了模块的相互联系,使模块可作为插件或积木使用,降低程序的复杂性,提高了程序可靠性。如图 1-1 所示为一个典型的采用结构化程序设计的程序结构图。

结构化程序相比于非结构化程序有较好的可靠性、易验证性和可修改性;结构化设计方法的设计思想清晰,符合人们处理问题的习惯,易学易用、模块层次分明,便于分工开发和调试,且程序

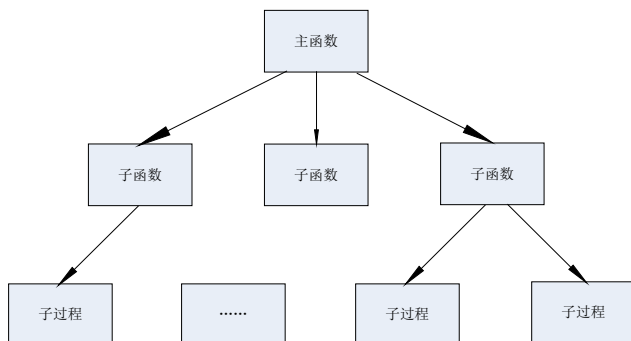


图 1-1 结构化程序设计

可读性强, 其设计语言有 Ada、Basic 等语言。

1.2.2 面向对象程序设计

20 世纪 70 年代中后期, 由于结构化程序设计在进行大型项目设计时存在缺陷, 于是面向对象程序设计方法 (Object Oriented Programming, OOP) 被提出, 并逐步代替了传统的结构化程序设计方法, 成为最重要的方法之一。

面向对象认为世界是由各种对象组成的, 任何事物都是对象, 复杂的对象可由较简单的对象以某种方式组成。由于面向对象的这种思想符合人们认识世界的观念, 因此, 面向对象程序设计一经提出, 就得到了广泛的支持, 至今已应用在各个领域。

面向对象程序设计方法是以“对象”为中心进行分析和设计的, 使这些对象形成了解决目标问题的基本构件, 即解决从“做什么”到“怎么做”的问题。其解决过程从总体上说是采用自低向上的方法, 先将问题空间划分为一系列对象的集合, 再将对象集合进行分类抽象, 一些具有相同属性行为的对象被抽象为一个类, 类还可抽象分为子类、超类 (超类是子类的抽象)。采用继承来建立这些类之间的联系, 形成结构层次。



提示 采用面向对象程序设计方法 (即 OOP) 来进行程序设计, 其本质上就是不断设计新的类示和创建对象的过程。

如图 1-2 所示, 为一个类和对象的关系示意图。

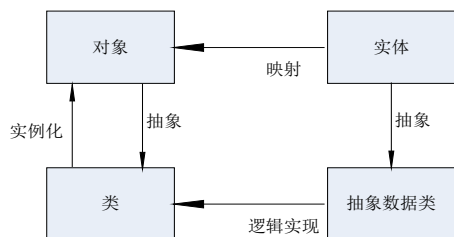


图 1-2 类和对象关系示意图

与传统的结构化程序设计方法相比, 面向对象程序设计方法具有许多优点:

- 采用对象为中心的设计方式, 再现了人类认识事物的思维方式和解决问题的工作方式。
- 以对象为唯一的语义模型, 整个软件任务是通过各对象 (类) 之间相互传递消息的手段协同完成的。因此, 能尽量逼真地模拟客观世界及其事物。

- 由于对象和类实现了模块化, 类继承实现了抽象对象, 以及任一对象的内部状态和功能的实现的细节对外都是不可见的, 因此很好地实现了信息隐藏。由此建立在类及其继承性基础上的重用能力可应付复杂的大型软件开发。

面向对象方法使得软件具有良好的体系结构、便于软件构件化、软件复用和良好的扩展性和维护性, 抽象程度高, 因而具有较高的生产效率。目前, 面向对象程序设计语言主要有 Java、C++ 等语言。

1.2.3 程序设计方法比较

通过 1.2.1 节和 1.2.2 节读者可以知道, 目前流行的程序设计方法有很多, 但真正具有广泛意义的是结构化程序设计、面向对象程序设计, 以及 20 世纪 90 年代后逐渐发展起来的基于构件的程序设计方法。这些方法各有自己的特点, 面向对象技术和构件技术是目前程序设计领域的热点, 但是, 结构化程序设计方法对面向对象程序设计中每个小模块 (即成员函数) 的设计也起到关键作用, 两者各有特点。

- 结构化设计方法的设计思想清晰, 易学易用, 模块层次分明, 便于分工开发和调试, 编写出来的程序可读性强。
- 面向对象设计方法具有自下而上的特性, 允许开发者从问题的局部开始, 在开发过程



中逐步加深对系统的理解。

由于面向对象程序设计是一种自下而上的程序设计方法，其不像过程式设计那样一开始就要用主函数概括出整个程序，面向对象设计往往从问题的一部分着手，一点一点地构建出整个程序。面向对象设计以数据为中心，以类作为表现数据的工具，是划分程序的基本单位。而函数在面向对象设计中成为类的接口。

从理论上来说，对于设计阶段的输出，无论采用哪一种风格的设计方法，都可以用任何一种程序设计语言来编码实现，但实际上对于具体的任务和设计风格，总可以在众多的编程语言中挑选出一种最适合的，使用它能够在程序运行效率、开发效率、软件可维护性等方面达到令人满意的折中。

1.3 C++开发环境——Visual C++ 6.0

C++是一种语言，要使用C++进行程序开发，必须要有编译环境。目前市面上较为流行的C++编译器主要有 Borland 公司推出的 Borland C++和 Microsoft 公司推出的 Visual C++。鉴于易用性和通用性，本书使用的是 Visual C++ 6.0。本节将要介绍的是 Visual C++ 6.0 的集成开发环境（Integrated Development Environment，IDE），如图 1-3 所示即为该环境的主界面图。

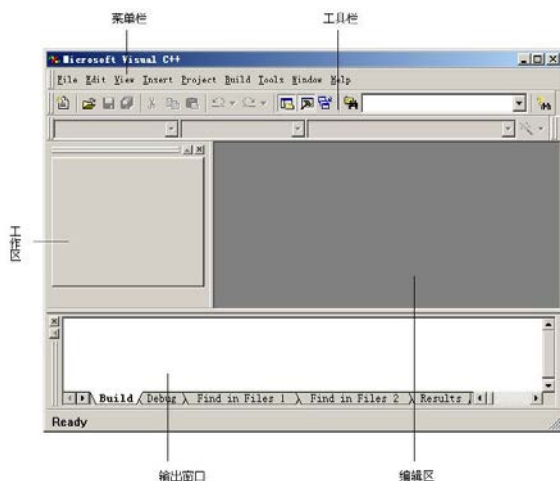


图 1-3 Visual C++ 6.0 集成开发环境



注意 本书中所有 C++源程序都是在 Visual C++ 6.0 的集成开发环境下编译运行的。

本节将详细讲解 Visual C++ 6.0 集成开发环境中的各个组成部分及其作用，以便读者在以后的使用中能熟练地使用该环境。

1.3.1 工作区

工作区（Workspace）窗口一般在集成开发环境的左侧区域，该区域在 Visual C++ 6.0 刚刚启动时不显示任何内容，当加载了某个工程或新建了一个工程的时候，工作区中就会以树形结构显示开发项目中的各部分内容，类似于 Windows 操作系统的资源管理器，如图 1-4 所示即为工程“Hello World”的工作区。

如图 1-4 所示，读者可以看到，工作区有三个图标标签，其分别



图 1-4 工作区窗口

允许用户以三种不同的方法查看应用程序的各个部分。

- **Class View (类视图):** 将工程中所包含的类、事件、函数及变量等在类视图图中以层次的结构排列, 不仅可以使用户快速找到它们, 并且可以直接双击它们, 以便用户在编辑区中操作源代码。
- **Resource View (资源视图):** 在 Visual C++ 中, 所有的菜单、图标、光标、图片、对话框等, 都是以资源的形式进行管理的, 而管理它们的就是 **Resource View**。在这里, 用户可以找到应用程序中的各种资源, 并且可以编辑它们的 ID 号及样式、属性等, 包括对话框的设计、图标、菜单等。
- **File View (文件视图):** 包含用户工程中的各种文件, 用户可以查看并编辑。



提示 当工作区没有显示在 Visual C++ 6.0 的集成开发环境中时, 可以通过单击【View】/【Workspace】菜单项来打开。

1.3.2 编辑区 (Editor Area)

编辑区 (Editor Area) 位于集成开发环境的右侧, 其是使用 Visual C++ 6.0 进行一切编辑的关键区域。在编辑 C++ 源代码时, 编辑区是代码编辑窗口; 在设计菜单、对话框或图片图标时, 编辑区是绘制窗口。如图 1-5 所示即是作为代码编辑窗口的编辑区。

总之, 对代码或资源的一切操作都将在编辑区中进行, 由于其重要性, 编辑区是不能被窗口、菜单或工具栏所占据的。

1.3.3 输出窗口 (Output Panel)

第一次启动 Visual C++ 6.0 时, 也许看不到输出窗口 (Output Panel)。直至完成第一个应用的编译之后, 输出窗口就会自动出现在集成开发环境的底部。这时除非用户自己关闭它, 否则输出窗口将一直开着。输出窗口会给出多种对用户的提示信息, 主要包括如下三种:

- 编译程序的进展说明、警告及出错信息。
- 查找某个关键字所在的位置的信息。
- 在调试运行查看代码时, 用户所关心的所有变量的值等信息。

例如, 如图 1-6 所示即为在输出窗口中输出编译某个程序时, 返回的相关信息, 读者可以通过这些信息判断该应用程序是否有语法错误。

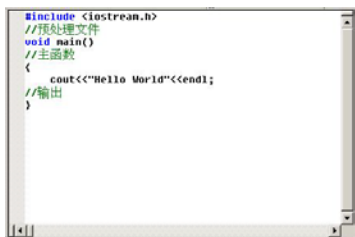


图 1-5 编辑区

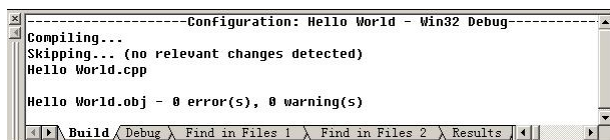


图 1-6 输出窗口



注意 当用户不小心将输出窗口关闭后, 该窗口将会在 Visual C++ 需要显示有关信息时, 自动打开并显示相关信息。



1.3.4 菜单栏、工具栏、状态栏

Visual C++ 6.0 的菜单栏 (Menu Bars) 中包含了多个菜单项, 每一个菜单项都对应着不同的功能, 通过系统菜单可以完成 Visual C++ 6.0 的所有功能。下面将简要介绍这些菜单及其对应的功能。

- **【File】菜单:** 主要提供工程及其文件的创建、打开、保存等功能, 其中**【Open/Save/Closes Workspace】**等菜单项是对整个工作区进行操作。
- **【Edit】菜单:** 主要给用户提供了便捷的编辑文件的手段, 如进行复制、粘贴、删除、查找等操作。在 Visual C++ 6.0 的**【Edit】**菜单中, 除了 Visual Studio 其他软件中类似的撤销、查找等功能外, 还增加了书签、高级、显示函数参数等菜单项。
- **【View】菜单:** 主要用来改变窗口的显示方式, 激活调试运行时所用的各个窗口。此外, 该菜单中还包括编辑类的向导 ClassWizard, 该工具使用频率极高。
- **【Insert】菜单:** 主要用于添加类、资源、文件、对象等到工程中。该菜单的菜单项比较少, 但在具体应用中的使用非常多。
- **【Project】菜单:** 主要用于添加文件到工程中并设置工程、导出生成文件等。工程 (Project) 是 Visual C++ 6.0 进行程序设计的基本单位, 因此该菜单也非常重要, 尤其是**【Add To Project】**菜单项用于为当前工程添加工程或文件, 其使用较多。
- **【Build】菜单:** 主要用于应用程序的编译、连接、调试和运行等。需要注意的是, 只有当工程或源程序经过编译后才能显示 Build 菜单的所有功能。例如, 当前新建了一个工程, Build 菜单中的 Execute 等菜单项则不会显示, 直到该工程通过编译才会显示。
- **【Tools】菜单:** 主要用于选择或定制集成开发环境中的一些实用工具, 如组件管理工具、控件注册工具及各种浏览窗口等。
- **【Window】菜单:** 主要用于排列、打开、关闭集成开发环境中的各个窗口, 快速打开某源文件, 使窗口重新分离或组合等操作, 或者改变窗口的显示方式, 激活调用时所用的各个窗口。
- **【Help】菜单:** 以不同方式提供大量的帮助信息及浏览所有的键盘快捷方式, 该菜单中还提供了在 Web 上访问 Microsoft 所提供的联机帮助。



注意 除了上述菜单后, Visual C++ 6.0 在特定环境下还将增加某些菜单。例如, 当用户进入 Debug 调试环境时, 将增加**【Debug】**菜单。

工具栏 (Tool Bars) 以一组按钮的形式提供了操作菜单的快捷方式; 状态栏 (Status Bar) 以文本或进度条的形式显示应用程序目前的基本状态。

1.4 第一个 C++ 程序——Hello World

为了让读者更好地理解本章, 该节给出第一个 C++ 程序代码“Hello World”, 以及其在 Visual C++ 6.0 中的编译、连接和运行步骤。

1.4.1 创建源程序

C++ 源程序可以在 C++ 编译器——Visual C++ 中创建。打开 Visual C++ 6.0 的集成开发环境, 本书中采用的是 Visual C++ 6.0 Enterprise Edition, 即企业版, 如图 1-7 所示。

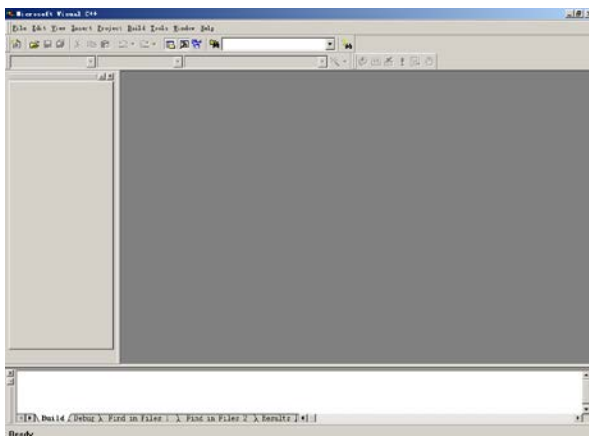


图 1-7 Visual C++ 6.0 集成开发环境

【范例 1-1】 第一个 C++ 程序——Hello World。该范例创建一个【C++ Source File】即 C++ 源程序文件，在其中输入相关代码，实现运行后在输出窗口中输出文本“Hello World”字样。其操作步骤如下：

- ① 单击菜单【File】/【New】，弹出如图 1-8 所示的对话框。
- ② 单击【File】选项卡，选择其中的【C++ Source File】项，并在右侧填写文件名及路径。在该示例中，文件名为“Hello World”，选择路径后，单击【OK】按钮，完成建立，如图 1-9 所示。

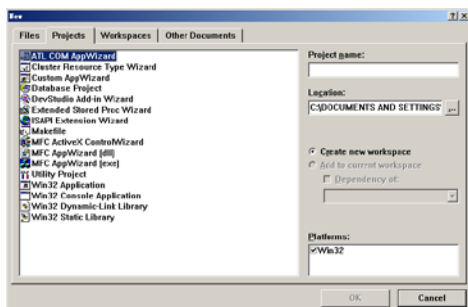


图 1-8 【新建】对话框

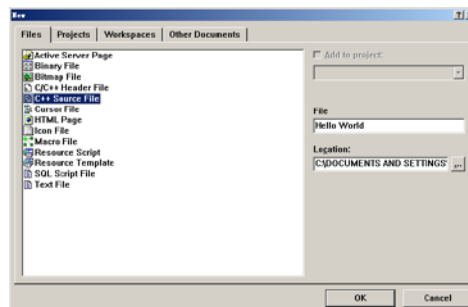


图 1-9 新建 C++ 源文件

提 在本章中，只需要用到该对话框中的文件页框。单击标签【File】，对话框显示如图 1-9 所示。

- ③ 在代码编辑框中输入实现输出的程序如代码清单 1-1 所示。

代码清单 1-1

```

1  #include <iostream.h>                                // 预处理文件
2  void main()                                           // 主函数
3  {
4      cout<<"Hello World"<<endl;                      // 输出
5  }
```

上述代码中，`#include <iostream.h>` 为运行该程序需包含的预处理文件。`main` 函数为主函数，由于该程序代码无返回值，因此在其前面加 `void` 关键字。`cout` 语句为输出语句，用于在运行后输出“Hello World”字符串，`endl` 为换行符。



至此，经过创建 C++ Source File 和在其中输入 C++源代码的步骤后，源程序的创建就完成了。下面需要进行编译连接和调试运行工作。

1.4.2 编译连接

创建上述源程序完成后，读者可以保存后使用 C++编译器对该源程序进行编译，以发现源程序中是否存在语法错误，编译完成后对其进行连接，以建立可执行文件。

在 C++中，执行源文件查看运行效果需要先编译连接该源文件。Visual C++ 6.0 中，使用菜单【Build】/【Compile】命令或快捷键【Ctrl+F7】进行编译，如图 1-10 所示；使用【Build】/【Build】菜单命令或快捷键【F7】进行连接，如图 1-11 所示。



图 1-10 编译



图 1-11 连接



注意 在编译连接过程中，如果源程序中有语法或连接错误，将不能通过编译连接，Visual C++ 系统在显示区给出详细错误信息，否则给出无错误信息，如图 1-12 所示为编译无错误下的显示，如图 1-13 所示为连接无错误下的显示。

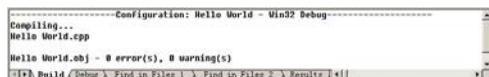


图 1-12 编译结果



图 1-13 连接结果

1.4.3 调试运行

当通过编译连接后，就可以运行该程序查看运行结果了。单击菜单【Build】/【Execute】项目或使用快捷键【Ctrl+F5】运行源程序，如图 1-14 所示。

【运行结果】运行程序后，系统给出一个命令提示符下的窗口显示运行结果。在该示例中，程序输出字符串“Hello World”，如图 1-15 所示。



图 1-14 运行

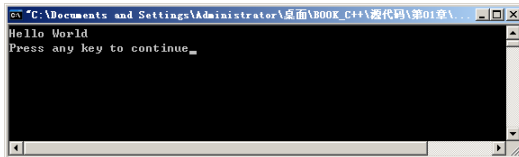


图 1-15 运行结果



警告 如果运行的输出结果与用户期望的不一致，那么就需要对该源程序进行功能调试，以找出逻辑上的错误。

至此,一个完整的 C++ 程序的建立及运行就完成了。在本章的所有示例中,建立、编译连接、运行源程序的步骤均与其类似。

1.5 C++源程序组成元素

在 C++ 语言的学习过程中,首先需要对 C++ 的基本组成、基本符号、标识符和保留字等有一定的了解,它们是阅读和编写程序的基础。

1.5.1 基本组成

一般来说,一个标准的 C++ 程序通常由预处理命令、函数、语句、变量、输入/输出及注释等几个部分组成。

- 预处理命令:在 C++ 程序中,预处理命令以“#”开始。C++ 提供三种预处理命令:宏定义命令、文件包含命令及条件编译命令。
- 函数:一个 C++ 程序通常由若干个函数组成,这些函数可以是 C++ 系统提供的库函数,也可以是用户根据需要编写的自定义函数。在这些函数中,必须有且仅有一个主函数 `main`,不论主函数位于什么位置,该程序都是从主函数开始执行的。
- 语句:语句是组成程序的基本单元,它包括顺序语句、选择语句、循环语句等。所有的语句以分号结束,最简单的语句是空语句,它仅包括一个分号。
- 变量:在 C++ 程序中,需要将数据存放于内存单元中,而变量就是用来存储和访问内存单元中数据的标识符。变量有整型、字符型、浮点型等基本数据类型。
- 输入/输出:在 C++ 程序中,经常要使用到输入和输出语句,用于接收用户的输入及返回程序运行结果。
- 注释:注释可以帮助读者阅读源程序,但并不参与程序的运行。

【范例 1-2】C++ 的基本组成。该范例体现了一个标准 C++ 程序的基本组成,其包含预处理命令、函数、语句、变量、输入/输出和注释等部分,代码如代码清单 1-2 所示。

代码清单 1-2

```
1  #include<iostream.h>           //预处理命令
2  void main(void)                //主函数
3  {
4  char name[10];                 //变量
5  cout<<"请输入姓名: ";         //输出
6  cin>>name[10];                //输入
7  cout<<"欢迎使用 Visual C++ 6.0"<<endl; //输出
8  }
```

【运行结果】同样在 Visual C++ 6.0 中新建一个 C++ Source File 文件后输入名称“1-2”,输入上述语句,编译并运行,其结果如图 1-16 所示。

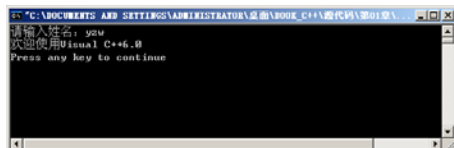


图 1-16 C++基本组成

【范例解析】上述程序代码使用到了 C++ 程序的所有组成部分,其中主函数 `main` 中带有“;”的均为语句,以“//”开头的均为注释。这是一个简单的 C++ 程序,其中就包含了上述基本程



序结构的所有组成部分，读者可仔细理解其各部分在程序中的作用。



事实上，C++有许多优点是C语言所不具备的，主要体现在封装性（Encapsulation）、继承性（Inheritance）和多态性（Polymorphism），这将在后续章节陆续讲解。

1.5.2 基本符号

每种语言都有自己的一套符号，符号是组成程序的基本单位，它是由若干字符组成的具有一定意义的最小单元，如标识符、关键字、运算符、分隔符、常量、注释符等。这里组成符号的字符必须是这种语言字符集中的合法字符，在C++中规定了一个自己的字符集，其组成词法的基本符号主要有以下三类。

- 字母：大小写英文字母：A~Z，a~z 共 52 个符号。
- 数字：数字字符：0~9 共 10 个符号。
- 特殊字符：空格、!、#、%、^、&、*、_（下划线）、+、=、-、~、<、>、/、\、'、"、;、:、,、()、[]、{}。

1.5.3 标识符

标识符是程序员定义的词法符号，用它来命名程序中的一些实体。常见的有函数名字、类名、变量名、常量名、对象名、标号名、类型名等。C++规定标识符由大小写字母、数字符号和下划线组成，并且第一个字符必须是字母和下划线。在C++中定义标识符需要遵循一定的规则，如下：

- 标识符长度没有限制，但不同的编译系统有不同的要求，一般不超过 31。
- 第一个字符必须是字母或下划线。
- 标识符中大小写是有区别的。XY，xy，xY，Xy 都是不同的标识符。
- 标识符定义时应尽可能使用有意义的单词。
- 标识符不能与关键字相同。
- 中间不能有空格。

一般来说，标识符的有效长度为 32，也就是说，长度超过 32 个字符的标识符，若前 32 个字符相同，那么认为是同一个标识符。



定义标识符时不能使用C++语言中的保留字，如if、for、int等都不能用做标识符。C++的保留字将在下面予以介绍。

1.5.4 保留字

在C++程序中，保留字又称为关键字，是有特定含义的单词。对于保留字，在编程时不能用于其他用途。表 1-1 中列出了常用的保留字，其含义和用法在相关的章节中再做详细介绍。

表 1-1 C++常用保留字

asm	auto	bool	break	case	catch	char
class	const	continue	default	delete	do	double
else	enum	extern	false	float	for	friend
goto	if	inline	int	long	main	namespace
new	operator	private	protected	public	register	Return
short	signed	sizeof	static	struct	switch	template

续表

this	throw	true	Try	typedef	typeid	typename
union	unsigned	using	virtual	void	volatile	while



需要注意的是,读者在定义标识符的时候,不能定义与表 1-1 中同名的标识符,否则在程序运行时将出现错误。

1.5.5 分隔符

C++语言中分隔符又称标点符号,用来分隔单词和程序正文。C++中常用的分隔符如下。

- 空格符: 用来做单词之间的分隔。
- 逗号: 变量说明时分隔多个变量。
- 分号: 作为语句结束时的标记。在 for 语句后面括号中三个表达式也用到分号。
- 冒号: 用做语句标号,在 switch 语句中也会使用到。
- {}: 用来构造程序。

最后简单介绍一下注释符。在 C++中,允许使用两种注释符,一种是 C++语言新增的注释方法,即以“//”开头,占一行。另外一种原来是 C 语言的注释方法,以一对“/*”和“*/”括起的注释信息。在本书中,将使用前一种注释方法,即以“//”引导注释。

1.6 小结

本章主要介绍了 C++语言的基础知识,从 C++语言的起源开始,简要介绍了 C++的特点、与 C 语言的区别、以 C++语言为代表的程序设计方法等知识。为使读者更好地理解本书源代码部分,重点讲解了本书使用到的 C++编译器——Visual C++ 6.0 的集成开发环境,并在该环境中编写了第一个 C++源程序,详细介绍了在其中建立源程序、编译连接、调试运行等步骤。此外,还简要介绍了 C++语言的组成部分,以便于读者分析。

1.7 习题

1. C++语言是 C 语言的扩充,其与 C 语言最大的区别是什么?

【解答】C++是一种面向对象的程序设计语言,而 C 是一种结构化程序设计语言,这两种编程思想的区别可以通过公式来表示,结构化程序设计的编程思想为:程序=算法+数据;面向对象编程思想为:程序=对象+事件。

2. 创建一个 C++程序的步骤主要包括哪些?

【解答】在 Visual C++中,创建一个 C++应用程序的主要步骤有:创建工程或源文件、输入编辑源程序、编译连接生成可执行文件、保存调试程序等步骤。

3. 创建一个 C++应用程序,要求其计算两个整数的和,并将结果输出。

【解答】该习题使用到了 C++的相关的基础知识和使用 Visual C++进行 C++程序的调试、编译和运行等步骤,具体操作如下:

① 创建一个【Win32 Console Application】工程,在【Project name】工程名中输入“求和”,在【Location】位置中选择工程的存储路径,如图 1-17 所示。

② 在图 1-17 中单击【OK】按钮后进入【Win32 Console Application—Step 1 of 1】对话框,在其中选择建立一个简单的应用程序【A simple application】,如图 1-18 所示。

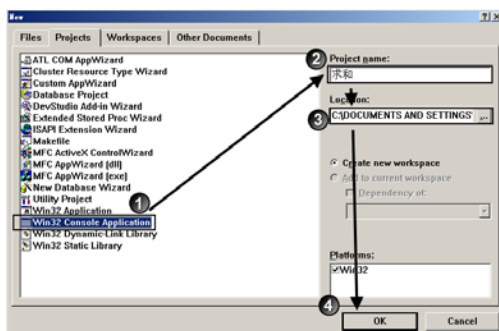


图 1-17 创建工程

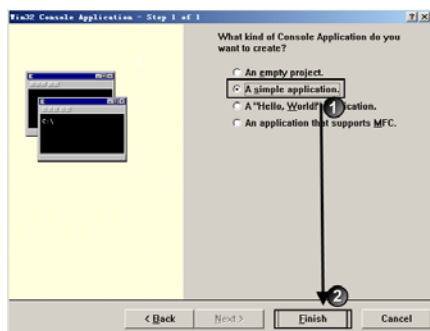


图 1-18 工程向导

提示 选择【A simple application】选项表示新建的工程已经含有一个 main() 主函数，尽管该函数没有任何有意义的代码。

③ 在图 1-18 中单击【Finish】按钮后，系统将给出创建工程的确认信息，其中列出了该工程的源程序文件和预编译头文件，如图 1-19 所示。

④ 在图 1-19 中单击【OK】按钮后就创建了一个终端应用程序，在集成开发环境 IDE 的左侧打开【FileView】页框，找到主程序“求和.cpp”，双击打开该文件，在右侧显示的即是代码编辑区，如图 1-20 所示。

提示 图 1-20 中右侧的代码编辑区中的代码是工程自动生成的，读者可在一对括号中的 return 语句前添加需要的代码。

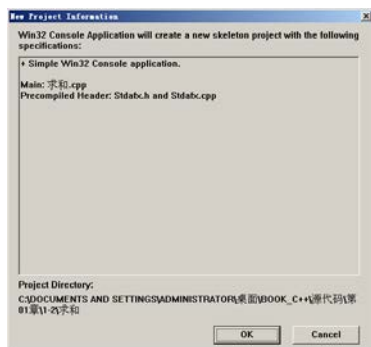


图 1-19 新工程信息

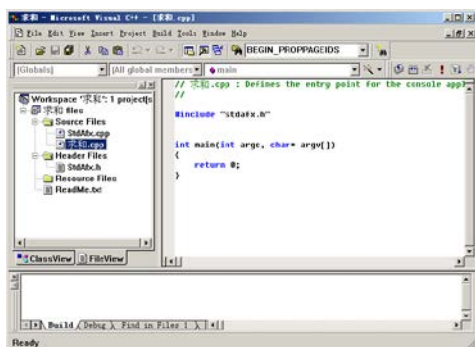


图 1-20 代码编辑区

⑤ 在代码编辑区中输入求两个整数相加的代码即可，如图 1-21 所示。

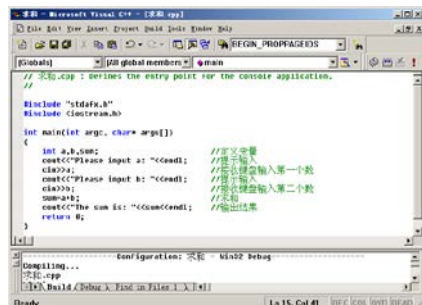


图 1-21 在编辑区中添加代码

⑥ 单击菜单命令【Build】/【Compile】命令或快捷键【Ctrl+F7】进行编译，使用【Build】/【Build】菜单命令或快捷键【F7】建立应用程序，就可以单击菜单【Build】/【Execute】项目或使用快捷键【Ctrl+F5】运行该程序了。运行该程序后，在其中输入 a 的值为 5，b 的值为 8，则其和为 13，运行结果如图 1-22 所示。

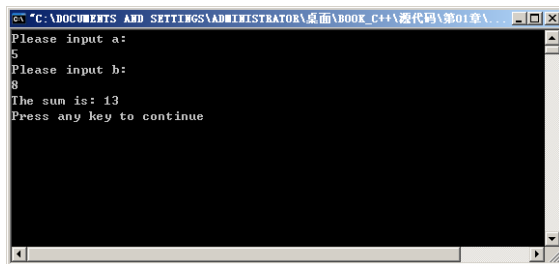


图 1-22 运行结果

该范例中通过建立【Win32 Console Application】工程来实现 C++源程序的功能，其创建步骤稍微复杂一些，使用到了 Visual C++的工程向导，在编译时也需要编译 stdafx.h 等头文件，并且主函数 main()带有参数和返回值。

4. 在 Visual C++中新建一个 C++源文件，并为该源文件命名为“main”，将该工程保存在“main”文件夹下。

【解答】C++源文件是 DEV-C++中输入编辑 C++源程序的文件，读者可以在 Visual C++集成开发环境中选择【New】菜单项，在弹出对话框中选择【File】标签，在其中选择“C++ Source File”后为其命名为“main”，并指定保存路径，如图 1-23 所示。

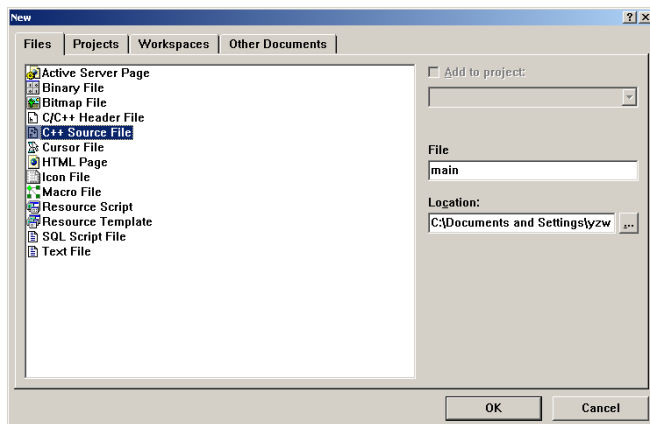


图 1-23 新建 C++源文件

5. 编写一个 C++程序，使得其运行时在用户屏幕上输出如图 1-24 所示的结果。

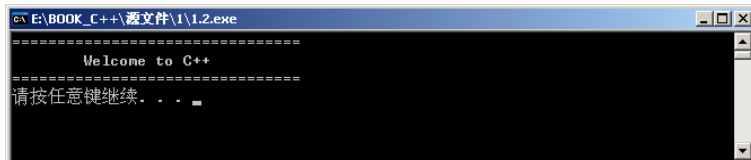


图 1-24 运行结果

【解答】该题要求程序运行后在用户屏幕输出 3 行字符，其中第 1 行和第 3 行字符为“=”



符号，中间字符为“Welcome to C++”。在第一个 C++ 程序“Hello World”中提到了使用输出流 `cout` 在用户屏幕上输出字符，因此此处同样可以实现。实现代码如下所示：

```
#include <iostream.h>
main()
{
    cout<<"===== "<<endl;
    cout<<"      Welcome to C++      "<<endl;
    cout<<"===== "<<endl;
    return 0;
}
```

第 2 章 变量和数据类型

任何应用程序都需要处理数据，并需要一个地方来临时存储这些数据，这个存储数据的地方被称为内存。一般来说，内存中不同的位置可以通过唯一的地址来识别。早期的编程语言要求程序员以地址的形式跟踪每一个内存位置，以及保存在该位置的值。程序员使用该地址来访问或者改变内存的内容。随着编程语言的发展，通过引入变量这个概念得以简化对内存的访问。本章将就 C++ 中的变量和数据类型做详细介绍。

以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为一个学习的参照。

- 掌握 C++ 中的常量、变量及其定义。
- 掌握 C++ 中数据类型及其转换。
- 熟练掌握在 C++ 程序中如何声明及使用常量、变量和数据类型。

2.1 常量

C++ 程序中的数据可分为常量与变量两大类。常量是在程序运行过程中不变的量，变量是在程序运行过程中可发生变化的值。在编程时，常量和变量必须遵循“先声明，后使用”的原则，即所有常量和变量必须在使用前用说明语句进行说明，否则编译时将会产生错误。本节主要针对常量进行介绍，下一节将着重介绍变量的相关内容。

2.1.1 声明常量

一般来说，根据书写形式区分，常量可分为符号常量和直接常量。直接常量就是通常说的常数，如：123、3.14、“a”、“&”等。符号常量如：

```
const float pi=3.1415926;
```

上述声明就声明了常量 pi 为一个符号常量，在程序中用 pi 代替 3.1415926，提高了程序的可读性和可维护性。但是，在更多情况下，根据常量的定义方法区分，常量可分为如下两种：

- const 常量
- 宏常量

其中，用 const 定义的常量，称为正规常量，其说明语句的一般形式为：

```
const <类型名> <常量名> = <表达式>;
```

通常，使用 const 定义常量需要遵循一些注意事项，具体如下：

- 必须以 const 开头。
- 类型名为基本类型及其派生类型，可以省略。
- 常量名为标识符。
- 表达式应与常量类型一致。

而宏常量是用#define 定义的常量，其说明语句的一般形式为：

```
#define <宏名> <常量>
```

宏常量的具体定义规则如下：

- 宏名可以是简单的字符名，也可以是带有参数的函数名。



- 常量可以是数值、字符串和函数等。

例如，下列程序代码定义了常量 `pi` 的值为 3.1415926，分别采用 `const` 关键字和 `#define` 关键字定义，其定义语法稍有不同。

```
Const float pi=3.1415926;           //定义常量pi 的值为 3.1415926
#define pi 3.1415926;               //定义宏常量pi 的值为 3.1415926
```

上述程序代码定义符号常量 `pi`，然后在程序中使用 `pi`，跟使用常数 3.1415926 的效果是一样的。编译器在编译时，把符号 `pi` 替换成 3.1415926，当需要修改 `pi` 的值时，只需要修改上面的语句即可。



注意 预定义符号即定义宏常量与符号常量不同，在编译时使用常数替代了所有的预定义符号，这样在代码中相应位置实际都是常数。程序中过多的常数会导致程序代码量变大，而且在多个源文件中定义了同样的符号，会产生符号重定义的问题。使用常量优于 `#define` 宏，其优点在于可指定类型信息。

在程序设计中，尽量使用符号常量来代替常数，这是一种好的编程习惯，这样可以增加程序的可读性、可维护性。例如，在数值计算中，会经常遇到一些常量，比如圆周率。如果把它定义成符号常量，当需要更改常量值的时候，只需要更改符号常量的定义语句即可。

2.1.2 常量的应用

在进行程序设计时，常量的应用很普遍。例如，圆周率 $\pi=3.1416$ 。在需要常量的地方，直接使用常量的数值的方法非常不好。例如，需要计算圆的面积，采用下列表达式：

```
s = 3.1416*r*r;
```

然而，如果此处需要提高计算精度，将 π 的值改为 3.1415927 进行计算，用户就不得不将程序中所有的 π 值从 3.1416 改为 3.1415927，这不仅烦琐，更重要的是很容易出错。

【范例 2-1】 常量应用示例。该范例在一个 C++ 程序中定义了常量，并在主函数中使用该常量，读者可观察其使用方法和作用，其操作步骤如下所示：

① 创建 C++ 源程序。打开 Visual C++ 6.0 的集成开发环境，单击菜单 **【File】 / 【New】**，弹出 **【New】** 对话框。单击其 **【File】** 选项卡，选择其中的 **【C++ Source File】** 项，并在右侧填写文件名及路径。在该示例中，文件名为“2-1”，并选择该文件的保存路径，如图 2-1 所示。确定输入及选择的保存路径无误后，单击 **【OK】** 按钮，完成建立。

上述步骤中，单击 **【OK】** 按钮后，Visual C++ 将进入 C++ 源程序的编辑界面，如图 2-2 所示，用户可在该界面中输入源程序代码。

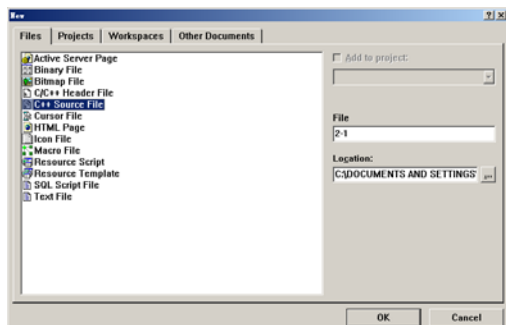


图 2-1 创建 C++ 源程序

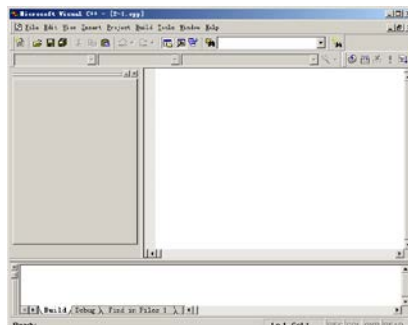


图 2-2 源程序编辑界面

- ② 在上述 Visual C++源程序的代码编辑区中输入源程序代码，如代码清单 2-1 所示。

代码清单 2-1

```

1  #include <iostream.h>                //预处理文件
2  void main()
3  {
4      const double pi=3.141592635898;    //定义圆周率常量pi
5      const double radius=8.5;          //定义半径常量radius
6      cout<<"area of circle of radius " <<radius <<" is "<<pi*radius*radius<<"\n";
7                                          //输出结果
8  }
```



注意 #include <iostream.h>为添加输入/输出库函数，这样在源程序中就可以使用诸如 cout 和 cin 等输入/输出函数，否则系统将报错。

- ③ 编译连接源程序。完成源程序的录入编辑后，需要查看该程序的运行效果，必须先对源程序进行编译和连接。在使用菜单【Build】/【Compile】命令或快捷键【Ctrl+F7】进行编译时，由于没有建立工作区，Visual C++将给出如图 2-3 所示的提示信息，询问用户是否建立一个默认的工作区。

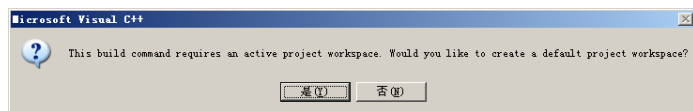


图 2-3 提示信息

此处单击【是】按钮即可，系统将在 C++源程序所在的目录下生成工程文件。如编译没有错误，那么使用【Build】/【Build】菜单命令或快捷键【F7】进行连接，建立应用程序，最终的编译连接结果应在输出窗口中显示，如图 2-4 所示。

【运行结果】经过上述步骤后，就可以运行该源程序，用以查看运行结果。该实例中定义的常量 pi 值为 3.141592635898，半径 radius 值为 8.5，其圆的面积为 $\pi \times \text{radius} \times \text{radius}$ ，因此，其运行结果如图 2-5 所示。

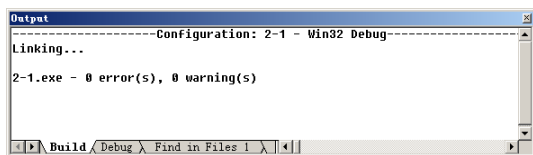


图 2-4 连接输出信息

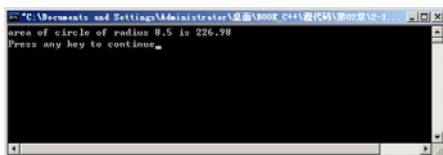


图 2-5 运行结果

【范例解析】读者可以看出，范例 2-1 的功能为打印给定半径的圆的面积。上述代码中，定义了两个符号常量，分别为 pi 和 radius，在定义中就给出了这两个常量的值，在最后的输出语句中给出求圆面积的公式即可。



提示 上述代码中定义的常量均为 double 型常量，表示双精度类型，这在下面的数据类型中将详细讲解。

至此，关于常量在具体程序中的应用就完成了。读者可以看到，在具体应用程序中，使用常量可以方便程序，但也造成一些问题，如上述示例中，半径的值只能在程序中指定，而不能根据用户的需求指定，这就限制了程序的使用范围，该问题的解决需要使用到下面要介绍的变量。



2.2 变量

与上一节介绍的常量只能存储不变的数据不同，变量用于存储一个可变数据，该数据的值可在应用程序中随时改变。在实际的应用程序中，变量的使用远多于常量，这是因为变量的使用更为灵活，且符合人们的思维习惯。

2.2.1 声明变量

一般来说，变量可以用来存储程序中需要处理的数据。在使用变量前，需要使用声明语句对变量进行声明。C++中变量说明语句的一般形式为：

[<存储类>] <类型名或类型定义> <变量名表>;

其中，存储类指的是变量的存储位置，一般来说有如下4种类型。

- **auto**: 属于一次性存储，其存储空间可被若干变量多次覆盖使用。
- **register**: 存放在通用寄存器中。
- **extern**: 在所有函数和程序段中都可引用。
- **static**: 在内存中是以固定地址存放的，在整个程序运行期间都有效。

其中，[<存储类>]的方括号表示可以默认，在默认情况下，变量的存储类值为 **auto**。读者可以根据不同需要声明不同的存储类变量，这在后续章节中还将提到。

类型名或类型定义指的是变量所属的数据类型，一般来说，指的是前面章节介绍的基本数据类型和枚举类型等。在任何变量说明语句中，数据类型定义不可默认。

变量名表是指声明变量的变量名称，此外，在变量的声明语句中，可以对该变量赋初值。因此，变量名表的格式主要有如下三种：

- <变量名>
- <变量名>=<表达式>
- <变量名 1>=[<表达式 1>], <变量名 2>=[<表达式 2>],...其中，表达式是指变量的初始值。例如，下面程序代码定义了几个变量。读者可参照注释仔细理解变量的声明方法，尤其是一条语句中声明多个变量的方法。如以下代码所示：

```
int a;                //声明整型变量 a
char b='A';           //声明字符型变量 b，并给其赋初值字符 A
float c=2.5,d,e=56.1; //声明浮点型变量 c,d,e，其中将 c 赋初值 2.5，e 赋初值 56.1
```

上述三条语句中，定义了5个变量，其中前两条语句各声明了一个变量，后一条语句声明了三个变量，并给其中的c和e赋了初值，这在应用程序中都是常见的声明方式。

2.2.2 变量的命名规则

C++中，变量的命名规则与其他高级语言的类似，读者如果有过高级语言诸如C语言等的学习经历，可以与之进行比较。

在C++中，变量是一种标识符，其命名规则必须遵循标识符的命名规则，即变量只能由大小写英文字母、下划线（_），以及阿拉伯数字组成，并且其第一个字符必须是大小写英文字母或者下划线，而不能是数字。例如，下列变量名为合法变量：**student**、**name2**、**s_age**、**_sno**等，而如下的这些变量名为非法的：

```
student$, studen]t, studen*    // $、] 和 * 都是非法字符
2name                          // 不能以数字开头
s-age                          // - 是非法字符
```



一般来说,操作系统和 C++ 标准库里的标识符一般以下画线开头,这是约定俗成的。因此,用户应该避免使用下画线作为自己定义的标识符的开头。此外,不能用关键字和保留标识符来给自定义的变量命名。



注意 C++ 语言是大小写敏感的语言,也就是说,star、Star、sTar、stAr 和 STAR 等都是相互不同的标识符。

事实上,变量的命名除了需要满足以上的命名规则外,为了保证程序的可读性,还需要对其进行一些约束。这是因为如果变量的命名毫无规律的话,那么程序员之间就不能看懂对方的代码写的是什么,在大型软件系统开发中,这将极大地影响开发效率。因此,好的命名规则意味着程序代码的易读性和高质量。

目前在 Windows 程序开发和 MFC 程序开发中常用的命名规则是匈牙利命名法。这种命名法的出发点是把变量名按“前缀+对象描述”的顺序组合起来,以使程序员命名变量时对变量的类型和其他属性有直观的了解,所有 Microsoft 的 API、界面、技术文件等都采用这些规范。应用匈牙利命名法,所有的变量名都应该以“前缀+名字”的形式出现,如表 2-1 所示给出了匈牙利命名法使用的前缀符号。

表 2-1 前缀符号表

前 缀	数据类型 (基本类型)
c	字符
by	字节
n	短整数和整数
i	整数
b	布尔型
w	无符号字
l	长整数
dw	无符号长整数
fn	函数指针
s	串
sz	以 0 为结束的字符串
lp	32 位长整数指针
h	句柄
msg	消息



提示 表 2-1 中,数据类型栏表示定义的变量为哪个类型,则在该变量前增加对应的前缀,表示该变量为某个类型。例如,下面变量就是符合匈牙利命名法的变量声明。

```
Char *szName;           //以 0 为结束符的字符串,存储的是名字变量
BOOL bCanExit;          //布尔型变量
DWORD dwMaxCount;       //32 位双字变量
```

2.2.3 变量的作用范围

在 C++ 语言中,声明的变量主要分为全局变量和局部变量,其可以出现在程序的任何位置,在不同的位置声明,其作用域不同。



- 全局变量：其说明语句不在任何一个类定义、函数定义或复合语句（程序块）中的变量。全局变量所占用的空间在内存的数据区，在程序运行的整个过程中位置保持不变。
- 局部变量：其说明语句在某一个类定义、函数定义或复合语句（程序块）中的变量。局部变量所占用的空间在为程序运行时所设置的临时工作区中，以堆栈的形式允许反复占用和释放。

【范例 2-2】变量的作用范围。范例 2-2 通过声明变量语句的位置，来确定全局变量和局部的作用域。

打开 Visual C++ 6.0 的集成开发环境，单击菜单【File】/【New】，弹出【New】对话框。单击其【File】选项卡，选择其中的【C++ Source File】项，并在右侧填写文件名及路径。在该示例中，文件名为“2-2”，并选择该文件的保存路径，如图 2-1 所示。

确定输入及选择的保存路径无误后，单击【OK】按钮完成建立源程序。系统进入 C++ 源程序的编辑界面后，在其中输入如代码清单 2-2 所示的代码。

代码清单 2-2

```
1  #include <iostream.h>           //预处理文件
2  void main()                     //主函数
3  {
4      int a=0;
5      int b=0;                     //定义全局变量 a,b,并赋初值 0
6      a++;
7      b++;                         //将 a 加 1 后赋给 a,将 b 加 1 后赋给 b
8      cout<<"a="<<a<<" "<<"b="<<b<<endl; //输出 a,b 的值
9      {
10         float a=0.5;             //定义局部变量 a
11         a++;
12         b++;                     //变量 a,b 递增
13         cout<<"a="<<a<<" "<<"b="<<b<<endl; //输出 a,b 的值
14     }
15     a++;
16     b++;                         //变量 a,b 递增
17     cout<<"a="<<a<<" "<<"b="<<b<<endl; //输出 a,b 的值
18 }
```

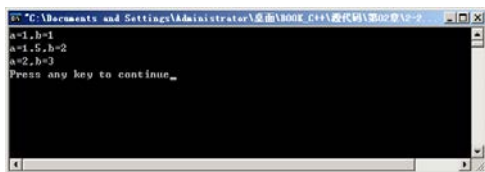


图 2-6 变量的作用范围

【运行结果】在 Visual C++ 6.0 中运行上述代码，其运行结果如图 2-6 所示。

【范例解析】在范例 2-2 代码段中，全局变量 int 型 a 的作用范围为 4~8 行和 15~17 行，全局变量 int 型 b 的作用范围为 5~17 行，而局部变量 float 型 a 的作用范围为 10~14 行。在上述代码中，主要使用了变量的声明语句和输出语句。



提 在实际的应用程序中，如果涉及全局变量，读者应仔细分析其中每个变量在程序中的作用范围及判断其值的变化。

2.2.4 变量的应用

为了更好地说明变量在具体程序中的应用，下面也通过一个实例对其进行讲解。在 2.1.2 节中介绍了计算圆面积的程序实现，但其半径只能在程序中指定，不能接收用户输入而根据输

入计算面积。下面对上述实例进行修改,使其达到接收用户输入而计算面积的目的。

【范例 2-3】根据用户输入计算圆面积。该范例根据用户输入圆的半径来计算其面积,其使用到了变量,操作步骤如下。

打开 Visual C++ 6.0 的集成开发环境,单击菜单【File】/【New】,弹出【New】对话框。单击其【File】选项卡,选择其中的【C++ Source File】项,并在右侧填写文件名及路径。在该示例中,文件名为“2-3”,并选择该文件的保存路径,如图 2-1 所示。

确定输入及选择的保存路径无误后,单击【OK】按钮完成建立源程序。系统进入 C++ 源程序的编辑界面后,在其中输入如代码清单 2-3 所示的代码。

代码清单 2-3

```

1  #include <iostream.h>                                //预处理文件
2  void main()
3  {
4      const double pi=3.141592635898;                //定义圆周率常量pi
5      double radius;                                  //定义半径变量radius
6      double area;                                    //定义面积变量area
7      cout<<"Please input radius"<<endl;
8      cin>>radius;                                    //输入半径
9      area=pi*radius*radius;                          //计算面积
10     cout<<"area of circle of radius " << radius << " is "<<area<<"\n";
11                                     //输出结果
12 }
```

【运行结果】运行上述程序后,输入半径 8.5,其运行结果如图 2-7 所示。

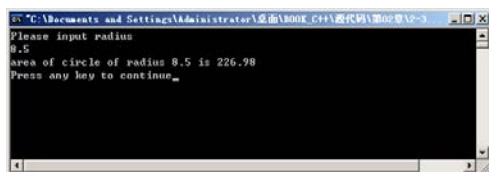


图 2-7 变量应用示例

【范例解析】范例 2-3 代码中,声明了两个变量 radius 和 area,其中 radius 用于接收用户输入的半径,area 用于存储计算得出的面积,最后输出。

注意 比较 2.1.2 节中的范例 2-1 与范例 2-3,读者可以发现,其不同点就在于前者使用常量,而后者使用变量,通过接收用户的输入来计算圆面积,这是较为实用的。

在具体的应用中,变量的使用非常广泛,几乎每个代码段中都需要使用到,在后续的实例中读者还将看到。

2.3 基本数据类型

数据类型是对系统中的实体的一种抽象,它描述了某种实体的基础特性,包括值的表示、存储空间的大小,以及对该值的操作。C++的数据类型包括基本数据类型和构造数据类型两类。构造数据类型又称复合数据类型,是一种更高级的抽象。当变量被定义为某种类型时会受到系统对该类型的特别保护,确保其值不受非法操作。为简单起见,此处只介绍 C++的基本数据类型,构造数据类型将在第 9 章中详细介绍。

一般来说,C++语言的基本数据类型有如下 4 种。



- 整型：说明符为 `int`。
- 字符型：说明符为 `char`。
- 浮点型（又称实型）：说明符为 `float`（单精度）、`double`（双精度）。
- 布尔型：说明符为 `bool`，只有两个取值。

为了满足各种情况的需要，上述的几种类型前面还可以加上修饰符改变原来的含义。主要的修饰符有如下 4 种。

- `signed`：表示有符号。
- `unsigned`：表示无符号。
- `long`：表示长型。
- `short`：表示短型。

上述 4 种修饰符都适用于整型和字符型，只有 `long` 还适用于双精度浮点型。数据类型规定的存储空间都是按字节算的，其占用的字节数会根据机器字长的不同而有所变化。



提示 一般来说，变量所占空间的大小与被定义的类型和机器有关，所以要注意哪些数据类型会受机器的影响。

C++中各种基本数据类型及其常用的派生类型的描述和取值范围如表 2-2 所示。

表 2-2 C++基本数据类型

数据类型	类型描述	占字节数	取值范围
<code>char</code>	字符型	1	-128~127
<code>unsigned char</code>	无符号字符型	1	0~255
<code>signed char</code>	有符号字符型	1	-128~127
<code>int</code>	整型	4	$-2^{31} \sim 2^{31}-1$
<code>unsigned [int]</code>	无符号整型	4	$0 \sim 2^{31}-1$
<code>short[int]</code>	短整型	2	-32768~32767
<code>unsigned short[int]</code>	无符号短整型	2	0~65535
<code>unsigned long[int]</code>	无符号长整型	4	$0 \sim 2^{31}-1$
<code>singed long[int]</code>	有符号长整型	4	$-2^{31} \sim 2^{31}-1$
<code>float</code>	单精度浮点型	4	-3.4e38~3.4e38
<code>double</code>	双精度浮点型	8	-1.7e308~1.7e308
<code>long double</code>	长双精度浮点型	10	-1.1e4932~1.1e4932
<code>void</code>	无值型	0	{}
<code>bool</code>	逻辑型	1	{false,true}

在上表中，如果在 `int` 整型数据类型前含有类型修饰符如 `short`、`signed` 等时，可省略 `int`。例如，`short int` 类型可简写为 `short` 类型。

2.3.1 整型

根据上面的介绍，读者可以知道，声明整型数据类型的关键字为 `int`，如果给其加上 `unsigned`、`singed`、`short` 和 `long` 等修饰符，整型数据类型可分为 4 种，分别对应为无符号整型、有符号整型、短整型和长整型。其分别对应的取值范围读者可参见表 2-2。例如，下列语句定义一个有符号的整型变量：

```
int i;
```

上述语句定义的变量 i 的取值范围为 $-2^{31} \sim 2^{31}-1$, 这在一般的应用中已经足够了。因此, 定义一个变量为 `int` 型是具体程序中应用较多的。除了定义整型变量外, 常量也有整型。在程序中书写整型常量时, 没有小数部分。用户可根据需要分别可以用十进制、八进制和十六进制的形式书写:

- 十进制格式: 由数字 0 至 9 和正、负号组成, 书写时直接写出数字, 如: 123, -516, +1000 等。
- 八进制格式: 以数字 0 开头的数字 (0 至 7) 序列, 如 0111, 010007, 0177777 等。
- 十六进制格式: 以 0x 或 0X 开头的数字序列, 如 0x78AC、0xFFFF 等。

【范例 2-4】整型数据类型的使用。该范例是一个使用了整型数据类型的程序, 在该程序中, 定义了一个整型变量, 给其赋值后输出, 实现代码如代码清单 2-4 所示。

代码清单 2-4

```

1  #include <iostream.h>           //包含输入/输出头文件
2  void main()
3  {
4      int i;                      //声明整型变量
5      i = 123;                   //赋值
6      cout<<i<<endl;            //输出
7  }
```

【运行结果】在 Visual C++ 中创建一个 **【C++ Source File】**, 将上述代码输入其中, 输出结果如图 2-8 所示。

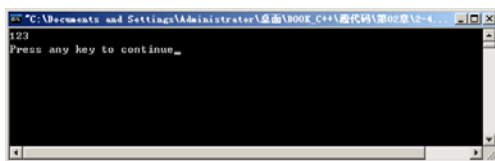


图 2-8 整型数据类型

【范例解析】范例 2-4 代码中, 首先声明整型变量 i , 接着为其赋值, 最后输出该变量中的值, 其中, `cout` 语句后的 `endl` 表示换行。



如果赋值时为该整型变量赋非整型的值, 如 123.4, 那么系统编译时将给出警告信息, 如图 2-9 所示。如果不理会该信息, 继续执行程序, 那么 Visual C++ 自动将小数去除, 只取整数部分输出。



图 2-9 警告信息

2.3.2 字符型

字符型数据类型只占据 1 个字节, 其声明关键字为 `char`。同样, 可以给它加上 `unsigned`、`singed` 修饰符, 分别表示无符号字符型和有符号字符型。例如, 下列语句定义一个有符号的字符型变量:



```
char ch;
```

上述语句定义的变量 `ch` 可取任意 ASCII 码为 -128~127 的字符,在具体应用中使用较多的是 `ch` 中存储大小写字母。同样,声明字符型常量需要注意的是:用一对单引号括起来的一个字符,单引号只是字符与其他部分的分割符,不是字符的一部分,并且不能用双引号代替单引号。在单引号中的字符不能是单引号或反斜杠。例如:

```
'a' , 'A' , '#'           //合法的字符常量
'', '\', '\n              //非法的字符常量
"A"                       //不代表字符常量
```

另一种表示字符常量的方法是使用转义字符。C++规定,采用反斜杠后跟一个字母来代表一个控制字符,具有新的含义。此外,用一对双引号括起来的一个或多个字符的序列称为字符串常量或字符串。字符串以双引号为定界符,双引号不作为字符串的一部分。如:

```
"Hello", "Good Morning!", "I say: \" Goodbye!\""
```

字符串中的字符数称为该字符串的长度,在存储时,系统自动在字符串的末尾加以字符串结束标志,即转义字符 `'\0'`。

【范例 2-5】字符串数据类型的使用。该范例是一个使用了字符型数据类型的程序,在该程序中,定义了一个字符型变量,给其赋值后输出,实现代码如代码清单 2-5 所示。

代码清单 2-5

```
1  # include <iostream.h>
2  void main()
3  {
4      char ch1;           //声明字符型变量
5      ch1 = 'a';          //赋值
6      cout<<"ch1= " <<ch1<<endl;    //输出
7      int ch2;            //声明字符型变量
8      ch2 = 'a';          //赋值
9      cout<<"ch2= " <<ch2<<endl;    //输出
10 }
```

【运行结果】在 Visual C++中创建一个 **【C++ Source File】**,将上述代码输入其中,输出结果如图 2-10 所示。

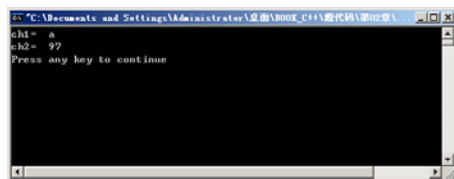


图 2-10 字符型数据类型

【范例解析】范例 2-5 代码中,定义了字符型数据 `ch1` 和整型数据 `ch2`,都给其赋值为字符“a”,输出后其结果不同,整型数据类型对应的变量 `ch2` 的输出为 97,这是因为字符型数据类型在计算机内部是转换为整型数据类型来操作的,如上述代码中的字母 a,系统会自动将其转换为对应的 ASCII 码值 97。



注意 大小写英文字母所对应的 ASCII 值是不一样的,小写字母对应的 ASCII 码值大于大写字母对应的 ASCII 值。

2.3.3 浮点型

浮点型数据类型亦即实数，当计算的表达式有精度要求时被使用。例如，计算平方根、正弦和余弦，它们的计算结果的精度要求使用浮点型。C++中有2种浮点型：单精度浮点型（float）及双精度（double）浮点型。其主要区别在于占用的字节数不同，前者为4个字节，后者为8个字节。

单精度浮点型（float）专指占用32位存储空间的双精度值。单精度在一些处理器上比双精度更快，而且只占用双精度一半的空间，但是当值很大或很小的时候，它将变得不精确。当用户需要小数部分并且对精度的要求不高时，单精度浮点型的变量是有用的。下面是一个声明单精度浮点型变量的例子：

```
float hightemp, lowtemp;
```

双精度型（double），正如其关键字“double”表示的，占用64位的存储空间。当用户需要保持多次反复迭代的计算的精确性时，或在操作值很大的数字时，双精度型是最好的选择。例如，前面计算圆面积，声明的常量和变量均为双精度型，如下所示：

```
double radius, area;
```

【范例 2-6】浮点型数据类型的使用。该范例是一个使用了浮点型数据类型的程序，在该程序中，定义了一个双精度型变量，给其赋值后输出，实现代码如代码清单 2-6 所示。

代码清单 2-6

```
1  # include <iostream.h>
2  void main()
3  {
4      double a;                //声明双精度浮点型变量
5      a= 3.13564824;           //赋值
6      cout<<a<<endl;          //输出
7  }
```

【运行结果】在 Visual C++ 中创建一个 **【C++ Source File】**，将上述代码输入其中，输出结果如图 2-11 所示。

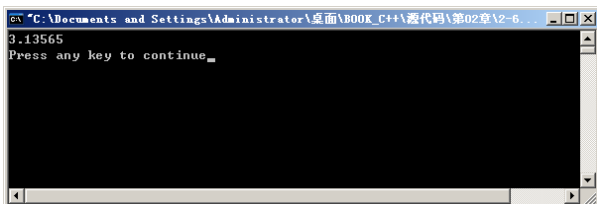


图 2-11 浮点型数据类型

【范例解析】范例 2-6 代码定义了一个双精度浮点型变量 a，并给其赋值后输出。读者可以看到，浮点型数据类型会对其精度进行取舍，如上述程序的输出并不会输出给定的变量 a 的完全值，而是进行了四舍五入。



提示 在 Visual C++ 6.0 中，使用较多的是 double 双精度数据类型。

2.3.4 布尔型

布尔型是最简单的数据类型，其只有两个值：true 和 false。同样，声明为布尔型的变量，是具有两种逻辑状态的变量，包含两个值：真和假。

此外，如果要把一个整型变量转换成布尔型变量时，其对应关系如下：



- 如果整型值为 0，则其布尔型值为假（false）。
- 如果整型值为 1，则其布尔型值为真（true）。

【范例 2-7】布尔型数据类型的使用。该范例是一个布尔型数据类型的程序，在该程序中，定义了一个布尔型变量，并为其赋值 true，读者可观察其作用，实现代码如代码清单 2-7 所示。

代码清单 2-7

```
1  # include <iostream.h>
2  void main()
3  {
4      double flag;                //声明布尔型变量
5      flag= true;                 //赋值
6      cout<<flag<<endl;         //输出
7  }
```

【运行结果】运行上述代码后，其结果如图 2-12 所示。

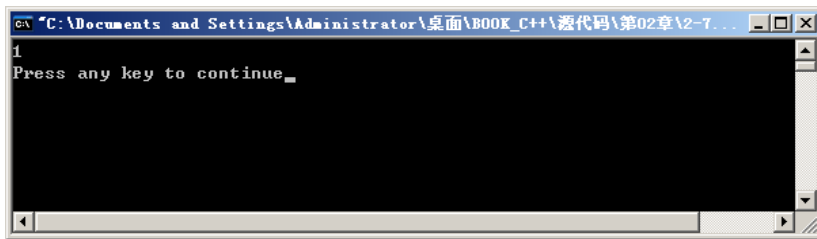


图 2-12 布尔数据类型

【范例解析】范例 2-7 代码定义了布尔型变量 flag，并给其赋值后输出。读者可以看到，其输出并不是 true，而是输出整数 1，这是使用布尔数据类型需要注意的。



提示 如果在算术表达式中使用布尔型变量，那么将根据变量值的真假而赋予整型 1 或 0。

除了上述介绍的 4 种基本数据类型外，C++还支持空值型（void）数据类型，这里就不再细述了，有兴趣的读者可参考相关文献。

2.4 类型转换

类型转换是用来把一个类型的值转换成另一个类型。C++是强类型的语言，即其每一个值都有它相应的类型。当用户需要把一个值转换为另一个类型时，就需要使用一些方式进行类型转换。C++中，支持隐式转换和显式转换两种。

2.4.1 隐式转换

隐式转换就是系统默认的，不需要加以声明就可以进行的转换。在隐式转换过程中，编译器无须对转换进行详细检查就能够安全地执行转换。比如从 int 类型转换到 long 类型就是一种隐式转换。隐式转换一般不会失败，转换过程中也不会导致信息丢失。

【范例 2-8】隐式转换的实现。该范例实现数据类型的隐式转换，将字符型变量转换为整型变量，实现代码如代码清单 2-8 所示。

代码清单 2-8

```
1  #include <iostream.h>
```

```

2   void main()
3   {
4       char ch;                //声明字符型变量
5       ch='a';
6       int i=ch;               //声明整型变量并赋初值，隐式转换
7       cout<<ch<<endl;        //输出
8       cout<<i<<endl;
9   }

```

【运行结果】在 Visual C++ 中创建一个【C++ Source File】，将上述代码输入其中，输出结果如图 2-13 所示。

【范例解析】在范例 2-8 程序中，声明了一个字符型变量 `ch` 和一个整型变量 `i`，并在声明变量 `i` 的同时给其赋初值 `ch` 的值，此处就需要将字符型的值转换为整型值，Visual C++ 编译器自动实现了其之间的转换，称为隐式转换。

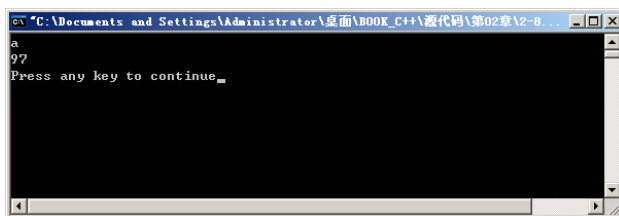


图 2-13 隐式转换



提示 图 2-13 中输出整型变量 `i` 的值为 97，这是因为字母 `a` 对应的 ASCII 码值为 97。由此可看出，字符型和整型数据类型之间可以相互转换。

2.4.2 显式转换

通常，隐式转换意味着编译器认为转换是合理的或者是安全的。此外，C++ 还支持显式转换，显式转换是用户手动指出需要转换的类型。显式转换意味着编译器能够找到一个转换方式，但是它不保证这个转换是否安全，所以需要程序员额外指出。

C++ 的显式转换提供了更精确的语义和对其进一步扩展的可能。在 C++ 语言中，读者可以用一个简单的 (int) 来完成 2.4.1 节中字符型到整型的转换。

【范例 2-9】显式转换的实现。该范例实现了数据类型之间的显式转换，将字符型变量通过 (int) 符显式转换为整型变量，实现代码如代码清单 2-9 所示。

代码清单 2-9

```

1   #include <iostream.h>
2   void main()
3   {
4       char ch;                //声明字符型变量
5       ch='a';
6       int i;                  //声明整型变量并赋初值
7       i=(int)ch;              //显式转换
8       cout<<ch<<endl;        //输出
9       cout<<i<<endl;
10  }

```

【运行结果】在 Visual C++ 中创建一个【C++ Source File】，将上述代码输入其中，输出结果如图 2-14 所示。

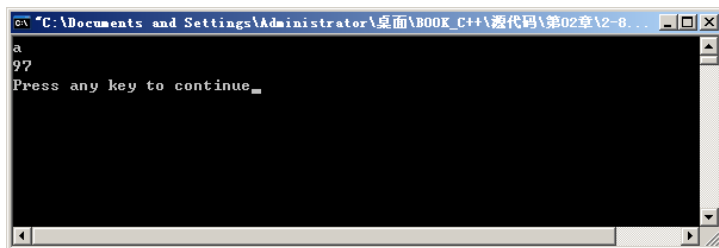


图 2-14 显式转换

【范例解析】在范例 2-9 代码中，使用了语句“`i=(int)ch;`”将字符型变量 `ch` 的类型强制转换为整型，因此，其运行后的效果与图 2-13 所示相同。事实上，C++ 中为显式类型转换提供了四种不同的操作符：`static_cast`、`dynamic_cast`、`const_cast`、`reinterpret_cast`。



注意 C++ 的显式转换通过区分各种转换情况来增加安全性：通过 `const_cast` 来取消 `const`、`volatile` 之类的修饰，通过 `static_cast` 来做相关类型的转换，通过 `reinterpret_cast` 来做低级的转换等。

2.5 小结

本章主要介绍了 C++ 基础的常量、变量和基本数据类型。常量、变量和基本数据类型都是计算机语言最基础的部分，读者需仔细理解其基本概念，在以后的程序设计中才能运用自如。本章除了讲解常量和变量的概念及其声明方法外，还安排了应用示例以便读者更好地理解常量、变量在具体程序中的使用方法。对于基本数据类型，本章主要介绍了 4 种基本类型：整型、字符型、浮点型和布尔型，针对每种类型，都使用了一个示例讲解其具体应用。最后简要介绍了 C++ 中类型转换的两种方式：隐式转换和显式转换。

2.6 习题

1. C++ 中如何声明变量，在声明时需要注意哪些事项？

【解答】C++ 可以随时定义所需的变量，而不必放在函数的开始处。定义变量时，先指定变量的类型，再给出变量名，并以分号“`;`”作为结束。例如，`int a=1;` 即声明了一个整型变量。

2. 编写一个 C++ 程序，根据用户输入的圆半径计算圆面积，并设定圆周率 `pi` 为 3.14，将运算结果在用户屏幕输出，如输入圆半径 5，其计算结果如图 2-15 所示。

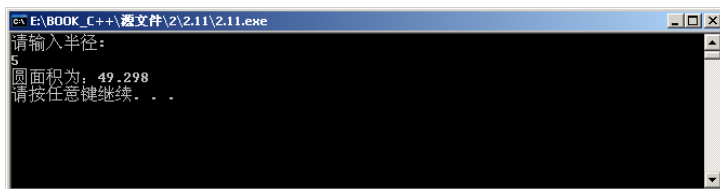


图 2-15 计算圆面积

【解答】设定圆周率为 3.14，因此需要声明圆周率为常量，根据本章节讲解内容，可声明其为符号常量。此外，需要接收用户输入圆半径，因此需要声明一个变量用于接收输入。同时将圆面积的计算结果以变量形式输出，因此需声明 2 个变量。其中涉及的所有常量和变量的数据类型都应该为浮点型，可以是单精度型，也可以是双精度型。其简要代码如下：

```
const double pi=3.14;
```

```
double radius;
double cir;
cout<<"请输入半径: ";
cin>>radius;
cir=pi*radius*radius;
cout<<"圆积为: "<<cir<<endl;
```

3. C++中, 如何声明常量, 包括直接常量和符号常量的声明?

【解答】C++中的常量包括直接常量、符号常量和枚举型常量, 其中直接常量通过具体常数来表示, 而符号常量通过关键字 `const` 来声明, 符号常量必须有一个常量名, 其必须符合 C++ 的标识符命名规则, 如 `const int a=1` 即声明了一个符号常量。

4. 字符串常量“C++”的字符个数是多少?

【解答】字符数据类型中都是以 `char` 进行变量定义, 一个字符变量只能包含一个字符。而字符串是一种特殊的字符数组, 其与字符的区别在于字符串都是以 ‘\0’ 结束的。因此, 字符串“C++”的字符个数为 4, 即包含 C、+、+这 3 个字符和 ‘\0’ 字符。

5. 整型变量 a 定义后赋初值的结果是什么?

```
int a=5+2+1
```

【分析】在定义整型变量并同时为其赋值后, 该变量的值为赋值号右边表达式的计算结果。因此, 上述语句变量 a 的值为 8。

6. 要实现根据用户输入的 x 值, 计算函数 y 的值, 函数 y 的值定义如下:

- 当 x 大于某一个数 10 时, $y = M * x + 1$
- 当 x 小于某一个数 10 时, $y = (x + M) * x - 3$

【解答】此处可定义符号常量 M 的值为 -1, 定义了整型常量 N 的值为 10, 定义了变量 x 和 y, 分别用于接收用户输入和输出结果。此外, 该范例中还需将变量与常量进行比较, 使用分支语句。实现代码如下所示。

```
#include<iostream.h>
#define M -1 //符号常量中的字母通常采用大写
const int N=10; //定义常量
void main()
{
    int x,y; //定义变量
    cout<<"请输入一个整数:";
    cin>>x; //接收输入
    if(x<N) //比较大小, x<N 成立
        y=M*x+1; //执行该语句
    else //x<N 不成立
        y=(x+M)*x-3; //执行该语句
    cout<<x<<" "<<y<<endl; //输出结果
}
```

7. 以下程序段的输出结果是多少?

```
#include <iostream.h>
void main()
{
    bool b;
    int i;
    b=true;
    i=0;
    cout<<"b="<<(int)b<<endl;
    cout<<"i="<<(bool)i<<endl;
}
```




【解答】该程序段输出显式类型转换的值，将布尔型变量 `b` 以整型值输出，将整型变量 `i` 以布尔值输出。其中，布尔型变量 `b` 的初始值为 `true`，整型变量 `i` 的初始值为 `0`，对其进行类型转换。变量 `b` 转换后的值为 `1`，这是因为 `true` 的值对应为 `1`，而变量 `i` 转换后的输出仍为 `0`，这是因为布尔型数值 `false` 的值对应为 `0`。因此，输出结果应为 `1` 和 `0`。

8. 以下程序段输出的结果是多少？

```
#include <iostream.h>
void main()
{
    char ch='c';
    int a;
    a=ch;
    cout<<a<<endl;
}
```

【解答】该程序段声明了字符型变量 `ch` 和整型变量 `a`，并将字符型变量的值字符 `A` 赋值给整型变量 `a`。由于 `255` 以下的整型数值与字符型变量是能够相互转换的，其基于 `ASCII` 码进行转换。因此，输出值应为字符 `c` 的 `ASCII` 码值：`99`。

第 3 章 运算符和表达式

运算符和表达式是程序设计语言的基本组成部分。C++语言中运算符是表示实现某种运算的符号，表达式是运算符和操作数的组合，通过运算符和表达式可实现程序编制中所需的大量操作。本章将介绍 C++的运算符的类型、优先级、结合规则及表达式等基本内容，并且变量和数据类型将作详细介绍。以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为学习的参照。

- 掌握 C++支持的各种运算符及应用。
- 掌握 C++支持的由各种运算符和常量变量构成的表达式，语句及其应用。

3.1 运算符

C++中包含了 C 语言中的运算符和表达式，并且又增加了一些新的运算符。如下：

- `::`作用域运算符
- `new` 动态分配内存单元运算符
- `delete` 删除动态分配的内存单元运算符
- `*`和`→`成员指针选择运算符

C++语言中的运算符是可以让 C++语言编译器能够识别的具有运算意义的符号。编译器把这些符号及其组成的表达式翻译成相应的机器代码后，就可以由计算机运行得出正确的结果。比如，日常生活当中许多东西的名字，如冰箱、电视机等分别代表不同功能的电器设备，那么运算符就是代表 C++语言中的各个运算功能的名字，这些名字是由制定 C++语言规范的人员确定的。

除了上述提到的一些新的运算符，C++提供的基本运算符有以下几种：算术运算符、关系运算符、逻辑运算符、位运算符、条件运算符、赋值运算符、逗号运算符、`sizeof` 运算符及其他运算符（按功能分）。不同的运算符需要指定的操作数的个数并不相同。根据运算符需要的操作数的个数，可将其分为 3 种：单目运算符（一个操作数）、双目运算符（两个操作数）和三目运算符（三个操作数）。下面，介绍几种基本的 C++运算符。

3.1.1 算术运算符

C++的算术运算符包含单目运算符和双目运算符，其中单目运算符有减、增量、减量运算符，双目运算符有加、减、乘、除和模运算符。C++语言中支持的算术运算符符号、名称、功能及其相关示例，如表 3-1 所示。

表 3-1 算术运算符

运 算 符	运算符名称	功 能	实 例	结 果
+	加法运算符	表示两个数相加	a+b	14
-	减法运算符	表示两个数相减	a-b	6
*	乘法运算符	表示两个数相乘	a*b	40
/	除法运算符	表示两个数相除	a/b	2.5
%	模运算符	表示取模	a%b	2



续表

运算符	运算符名称	功能	实例	结果
++	增量运算符	表示数自身加 1	a++	11
--	减量运算符	表示数自身减 1	a--	9

其中，诸如加、减、乘、除和模等双目运算符是其他程序设计语言中也都包含的，其运算方法此处就不再赘述了。这里对单目运算符做一个简单介绍，主要有减、增量、减量三种单目运算符，其功能如下。

- 单目减：单目减相当于负号，即对操作数取负号。
- 增量运算：增量运算有前缀增量和后缀增量两种形式。其中前缀增量的一般形式为：

++<运算分量>;

表示使用运算分量前其值加 1。后缀增量的一般形式为：

<运算分量>++;

表示使用运算分量后其值加 1。

- 减量运算：减量运算除了将加法改为减法外，其他和增量运算完全相同。

在进行包含多个算术运算符的表达式运算时，需要注意算术运算符的优先级。针对表 3-1 所示的算术运算符，其优先级如下：

单目运算符>*或/>%>+或-



注意 读者要注意算术运算符的结合性，运算符的结合性是指运算符和操作数的结合方式，其有“从左到右”和“从右到左”两种。

对于优先级相同的运算符，按照其结合性进行处理；在算术运算符中，除单目运算符外，其余双目运算符的结合性都是从左到右的。

【范例 3-1】算术运算符的使用。该范例中 a=5，b=3，c=1，d=2，e=6，求表达式 a+b-c/d+e%d 的值，实现代码如代码清单 3-1 所示。

代码清单 3-1

1	#include <iostream.h>	//预处理文件
2	void main()	//主函数
3	{	
4	int a=5;	
5	int b=3;	
6	int c=1,d=2,e=6;	//定义操作数变量并赋初值
7	int res=0;	//定义存放运算结果的变量
8	res=a+b-c/d+e%d;	//将运算结果存放在变量 res 中
9	cout<<" 运算结果为: "<<res<<endl;	//输出运算结果
10	}	

【运行结果】与前面章节提到的一样，经过编译连接该源程序没有错误后，使用菜单**【Build】**/**【Execute】**或快捷键**【Ctrl+F5】**运行，其结果如图 3-1 所示。

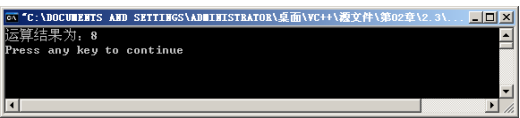


图 3-1 算术运算运行结果

【范例解析】根据算术运算符的优先级和结合性，可以计算得出 a+b-c/d+e%d=5+3-0+0=8。上述程序代码 res=a+b-c/d+e%d 将验证这一运算结果。在计算该表达式时，读者需要注意运算符的优先级，即乘法、除法和取模先进行

运算，加法和减法后运算。

3.1.2 赋值运算符

赋值运算符是 C++ 程序设计中最基本的运算符之一，利用赋值运算符可以给一个变量赋值。其说明语句的一般形式为：

<变量名>=<表达式>;

其中，各部分的作用如下：

- 表达式的类型需与变量的类型一致。
- “=” 为赋值运算符，其不同于数学上的等号。赋值运算的运算规则是先计算右边表达式的值，然后将值赋给左边的变量。

例如，代码清单 3-1 中的语句 “a=5” 或其他赋值语句，其中的 “=” 均为赋值运算符，而不是数学中的等号。此外，在程序中经常出现类似于 s=s+n 这样的赋值语句，C++ 允许采用更为简洁的形式写为 “s+=n”，于是形成了复合赋值运算符。根据 C++ 中算术运算符和比较运算符的种类，复合赋值运算符一共有 10 种，如表 3-2 所示。

表 3-2 复合赋值运算符

运 算 符	使用方法	等效形式	说 明
+=	a+=b	a=a+b	将 a 加 b 的值赋给 a
-=	a-=b	a=a-b	将 a 减 b 的值赋给 a
=	a=b	a=a*b	将 a 乘以 b 的值赋给 a
/=	a/=b	a=a/b	将 a 除以 b 的值赋给 a
%=	a%=b	a=(a%b)	将 a 除以 b 的余数的值赋给 a
<<=	a<<=b	a=(a<<b)	将 a 左移 b 位的值赋给 a
>>=	a>>=b	a=(a>>b)	将 a 右移 b 位的值赋给 a
&=	a&=b	a=(a&b)	将 a 与 b 逐位与的值赋给 a
=	a =b	a=(a b)	将 a 与 b 逐位或的值赋给 a
^=	a^=b	a=(a^b)	将 a 与 b 逐位异或的值赋给 a



说明 在赋值运算符中，复合赋值运算符和赋值运算符的优先级是相同的。赋值运算符的优先级仅高于逗号运算符。另外，赋值运算符的结合性是从右到左的。

【范例 3-2】赋值运算符的使用。下列程序代码实现求自然数 1~100 的算术和。在该示例中，使用到了赋值运算符和复合赋值运算符，实现代码如代码清单 3-2 所示。

代码清单 3-2

```

1  #include <iostream.h>           // 预处理文件
2  void main()                     // 主函数
3  {
4      int i,sum;                  // 定义变量
5      sum=0;                      // 变量 sum 用于存储结果，给其赋初值 0
6      for(i=1;i<=100;i++)        // For 循环
7      {
8          sum+=i;                 // 循环相加
9      }
10     cout<<"运算结果为: "<<sum<<endl; // 输出结果
11 }
```



【运行结果】编译连接上述源程序无误后，执行结果如图 3-2 所示。

【范例解析】范例 3-2 代码中使用了循环控制结构 for 语句，这将在第 4 章中详细讲解。此外，代码中使用了复合赋值运算符“+=”，并且语句“sum+=i;”相当于“sum=sum+i;”，即将变量 sum 本身加上变量 i 的值后将结果再赋值给 sum。

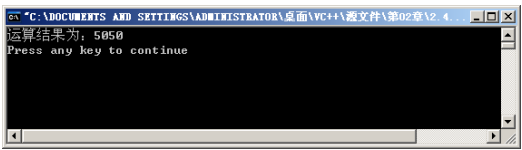


图 3-2 求自然数 1~100 和运算结果

提示 复合赋值运算符在实际程序中使用较多，其可以使得 C++语句更为精练、简单，读者应注意语句中变量值的变化。

3.1.3 关系运算符

关系运算符是双目运算符，其作用是将两个运算分量进行大小比较，其运算结果类型为布尔数据类型。若关系成立，则值为 true，否则为 false。C++中，支持的关系运算符主要有 6 种，其运算符号、名称、功能，以及各运算符的相关示例，如表 3-3 所示。

表 3-3 关系运算符

运 算 符	运算符名称	功 能	实 例	结 果
<	小于	若 a<b，结果为 true，否则为 false	2<3	true
<=	小于等于	若 a<=b，结果为 true，否则为 false	7<=3	false
>	大于	若 a>b，结果为 true，否则为 false	7>3	true
>=	大于或等于	若 a>=b，结果为 true，否则为 false	3>=3	true
==	等于	若 a==b，结果为 true，否则为 false	7==3	false
!=	不等于	若 a!=b，结果为 true，否则为 false	7!=3	true

上表中的前 4 种即<（小于）、<=（小于等于）、>（大于）、>=（大于或等于）优先级相同，它们的优先级高于==（等于）和!=（不等于）。此外，关系运算符的优先级低于算术运算符，其结合性是从左到右的。例如，下列语句中就包含了关系运算符，根据其关系成立与否，其返回值也不同。

```
int i=1,j=2,k;           //定义变量并初始化
k=(i>j);                 //关系运算
cout<<i<<endl;          //输出结果
```

3.1.4 逻辑运算符

C++提供了三种逻辑运算符，单目运算符有逻辑非(!)、双目运算符有逻辑与(&&)和逻辑或(||)。其运算结果类型为布尔型数据类型，其值为 true 或 false。这三种运算符号、名称、功能及其相关示例，如表 3-4 所示。

表 3-4 逻辑运算符

运 算 符	运算符名称	功 能	实 例	结 果
!	逻辑非	当运算分量为 false 时，结果为 true	!0	true
		当运算分量为 true 时，结果为 false	!1	false
&&	逻辑与	当两个运算分量都为 true 时，结果才为 true	0&&0	false
			0&&1	false
			1&&1	true



续表

运 算 符	运算符名称	功 能	实 例	结 果
	逻辑或	当两个运算分量有一个为 true 时, 结果就为 true	0 0 0 1 1 1	false true true

提示 表 3-4 中的!(逻辑非)、&&(逻辑与)、||(逻辑或)优先级依次从高到低。!(逻辑非)的优先级比算术运算符和关系运算符高, 而&&(逻辑与)、||(逻辑或)的优先级低于关系运算符。此外, 逻辑运算符的结合性是从左到右的。

例如, 下列语句包含了逻辑运算符及其相关计算。

```
int i=1, j=2, k; //定义变量并初始化
k=((i<j)&&(i>j)); //逻辑与运算
cout<<k<<endl; //输出结果
k=((i<j)|| (i>j)); //逻辑或运算
cout<<k<<endl; //输出结果
```

3.1.5 条件运算符

C++中, 还支持条件运算符的使用, 条件运算符是一个比较特殊的运算符, 其是三目运算符, 说明语句的一般形式为:

<表达式 1>? <表达式 2>: <表达式 3>

该表达式的使用规则如下。

- 表达式 1 必须是布尔类型。
- 表达式的执行顺序是: 先求解表达式 1; 若表达式 1 的值为 true, 则求解表达式 2, 表达式 2 的值为最终结果; 若表达式 1 的值为 false, 则求解表达式 3, 表达式 3 的值为最终结果。
- 条件运算符优先级高于赋值运算符, 低于逻辑运算符; 其结合性为从右到左。

例如, 读者可以通过如下的表达式理解条件运算符的使用。

```
x=a>b?a:b
```

上述表达式的含义为: 如果 a>b 成立, 那么将 a 的值赋给 x; 如果 a>b 不成立, 则将 b 的值赋给 x。这在实际程序中应用是比较多的。

【范例 3-3】条件运算符的使用。下列代码清单 3-3 根据用户输入的两个数值, 比较其值, 并输出其中大的数值, 实现代码如代码清单 3-3 所示。

代码清单 3-3

```
1  #include <iostream.h>
2  void main()
3  {
4      int a,b,x; //定义变量
5      cout<<"请输入两个数值: "<<endl;
6      cin>>a>>b; //接收用户输入
7      x=a>b?a:b; //条件运算符
8      cout<<"大的数值是: "<<x<<endl; //输出结果
9  }
```



【运行结果】编译连接上述源程序无误后，其执行结果如图 3-3 所示。

【范例解析】范例 3-3 程序代码中，使用了简单的输入/输出语句“cin”和“cout”，并使用条件运算符取得大的数值输出。其中，变量 x 的值根据 a 和 b 的大小来决定，如果 a>b，则 x 的值为 a，否则 x 的值为 b。

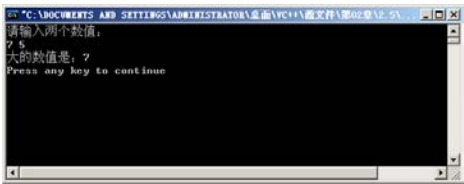


图 3-3 条件运算符执行结果

提示 事实上，读者在学习完后续章节中的控制语句流程内容后，可以发现，条件运算符其实相当于条件语句中的“if...else”语句结构。

例如，上述示例中的条件语句：x=a>b?a:b 可以通过“if...else”语句改写如下：

```
if a>b                                     //如果 a>b 成立
    x=a;
else                                       //如果 a<b
    x=b;
```

读者在学习条件控制语句时，可注意其写法。

3.1.6 逗号运算符

C++还支持逗号运算符的使用。逗号运算符可以使多个表达式写在一行上，从而大大地简化了程序，其一般形式为：

```
<表达式 1>,<表达式 2>
```

该表达式的使用规则如下。

- 表达式的执行顺序是：先求解表达式 1，再求解表达式 2，其最终结果为表达式 2 的值。
- 逗号运算符是优先级最低的运算符，其结合性为从左到右的。

例如，下列是一条采用了逗号运算符的语句：

```
x=2,x*5;
```

上述语句的结果为 10，这是因为逗号表达式的最终结果为后面的表达式值，而根据逗号表达式的求解顺序，其输出结果为 2×5=10。

3.1.7 位运算符

C++中除了提供对表达式的运算符外，还提供了对数据进行位运算的功能。C++中包含了支持数据位运算的 6 种位运算符，如表 3-5 所示。

表 3-5 中，左移运算(<<)指左移后，低位补 0，高位舍弃。右移运算(>>)指右移后，低位舍弃，高位无符号数补 0，有符号数补符号位。例如，下列语句对两个变量进行位运算，输出结果。

```
int i=1,j=2,k;                               //定义变量并初始化
k=i&j;                                         //按位与运算
cout<<k<<endl;                               //输出结果
```

表 3-5 位运算符

运 算 符	运算符名称	功 能	实 例	结 果
&	按位与	表示 a 与 b 按位与	二进制 1001&0101	二进制 0001
	按位或	表示 a 与 b 按位或	二进制 1001 0101	二进制 1101
^	按位异或	表示 a 与 b 按位异或	二进制 1001^0101	二进制 1100



续表

运 算 符	运算符名称	功 能	实 例	结 果
>>	右移位	表示 a 右移 b 位	二进制 1001>>2	二进制 0010
<<	左移位	表示 a 左移 b 位	二进制 1001<<1	二进制 0010
~	按位取反	表示 a 按位取反	二进制~1001	二进制 0110



注意 位运算符的优先级低于算术运算符，高于逻辑运算符&&（逻辑与）、||（逻辑或）。位运算符的结合性除单目运算符按位取反是从右到左，其余双目位运算符都为从左到右。

3.1.8 sizeof 运算符

由于不同的计算机支持的数据类型长度是不一样的，因此需要一个运算符来测量该机器中的数据类型长度。C++中提供了 sizeof 运算符，其就是用于测量类型长度的运算符。一般来说，该运算符的使用格式为：

```
sizeof(<类型名或表达式>)
```

该运算符的运算结果是类型名所表示类型的长度或表达式的值所占用的字节数，即这个值所属类型的长度。例如：

```
sizeof(int)=4
sizeof(double)=8
sizeof(100)=4
sizeof('a')=1
sizeof(struct ABC)           //求出结构类型 ABC 的长度
sizeof(a)                   //求出变量 a 的长度，亦即它所占用的字节数
```

3.1.9 运算符的优先级

3.1 节介绍运算符时，简要介绍了运算符的优先级，本节将着重介绍 C++的各种运算符之间及同一种类运算符的优先级顺序。

运算符优先级决定了在表达式中各个运算符执行的先后顺序。同一优先级的优先级别相同，运算次序由结合方向决定，使用括号可以改变运算符的顺序。例如表达式 1*2/3，*和/的优先级别相同，其结合方向自左向右，则该表达式等价于 (1*2)/3。C++中，运算符的优先级一般分为 15 级，其优先级中包含的运算符、功能说明和同一优先级的运算符结合性，具体如表 3-6 所示。



提示 此外，读者对运算符的结合方式也需有一些了解。运算符的结合方式是指优先级相同的运算符，其运算顺序由结合方式来决定。一般来说，运算符的结合方式有两种：左结合和右结合。大多数运算符都是从左到右计算，只有三类运算符的结合性是从右到左的，分别是：单目运算符、三目运算符和赋值运算符。

表 3-6 运算符优先级

优 先 级	运 算 符	功能说明	结 合 性
1	() :: [] .,-> . *,-> *	改变优先级 作用域运算符 数组下标 成员选择 成员指针选择	从左至右



续表

优 先 级	运 算 符	功能说明	结 合 性
2	++, -- & * ! ~ +,- () sizeof new, delete	增 1, 减 1 运算符 取地址 取内容 逻辑求反 按位求反 取正数, 取负数 强制类型 取所占内存字节数 动态存储分配	从右至左
3	*, /, %	乘法, 除法, 取余	从左至右
4	+, -	加法, 减法	
5	<<, >>	左移位, 右移位	
6	<, <=, >, >=	小于, 小于等于 大于, 大于等于	
7	==, !=	相等, 不等于	
8	&	按位与	
9	^	按位异或	
10		按位或	
11	&&	逻辑与	
12		逻辑或	
13	?:	三目运算符	从右至左
14	=, +=, -=, *=, /=, %=, &=, ^=, =, <<=, >>=	赋值运算符	从右至左
15	,	逗号运算符	从左至右

3.2 表达式

表达式是由运算符和操作数组成的式子，运算符可以是 3.1.9 节介绍过的各种运算符。操作数包含了常量、变量、函数和其他一些命名的标识符，最常见的表达式是常量和变量。此外，由于 C++ 中运算符很丰富，因此表达式的种类也很多。常见的表达式有如下 6 种：

- 算术表达式。例如，a+5.2/3.0-9%5
- 关系表达式。例如，'m'>='x'
- 逻辑表达式。例如，! a&&8||7
- 条件表达式。例如，a>4?a++:--a
- 赋值表达式。例如，a=7
- 逗号表达式。例如，a+5, a=7, a+=4

上述表达式的示例中，a 表示一个整型变量。除此之外，在具体的应用程序中使用表达式还需要注意如下事项：

- 编译系统将按尽量取大的原则来分割多个运算符。例如，表达式 a+++b;将被认为是 (a++) +b（注意：在 Visual C++ 中这种写法是错误的，编译将不能通过）。
- C++ 中可使用一对括号()来确定运算符组合。

C++中,使用运算符之前,有时需要确定其功能,因为有些运算符相同但功能不同。例如,运算符: *、&、-, 其有时是单目运算符,而有时是双目运算符。此外,表达式的运算顺序也是非常重要的,一般情况下,计算顺序由左至右,但其需要考虑到运算符的优先级和结合性。

表达式的类型由运算符的种类和操作符的类型决定。不同类型的表达式的求值方法和确定类型的方法也是不同的,下面的章节将逐一介绍。

3.2.1 算术表达式

算术表达式是由算术运算符和位操作运算符组成的表达式,其表达式的值是一个数值。表达式的类型具体由运算符和操作数决定。

【范例 3-4】算术表达式的应用。该范例使用到了较为复杂的算术表达式,其功能是计算该算术表达式的值,输出其结果和变量的变化情况,实现代码如代码清单 3-4 所示。

代码清单 3-4

```
1  #include<iostream.h>
2  void main()
3  {
4      int a,b,m=3,n=4;
5      a=7*2+-3%5-4/3;           //其顺序为先计算-3%5=-3, 4/3=1
6      b=(m++)-(--n);             //其顺序为先计算m++, --n, 再计算其差
7      cout<<a<<"\t"<<b<<"\t"<<m<<"\t"<<n<<endl;
8  }
```

【运行结果】编译执行上述代码,其执行结果如图 3-4 所示。

【范例解析】范例 3-4 代码中,根据算术运算符的优先级,计算算术表达式 $7*2+-3\%5-4/3$ 的值,并将其结果赋值给 a。此外,根据递增或递减运算符计算表达式 $(m++)-(--n)$ 的值并将其结果赋值给 b,最后将这些变量和结果输出。



警告 上述代码需要注意,语句 $b=(m++)-(--n);$ 在 Visual C++ 中执行必须使用括号将其运算顺序明确,否则将出现错误。

例如,如果写成如下格式:

```
b=m++--n;
```

那么在 Visual C++ 中编译时系统将会出现如图 3-5 所示的错误信息。

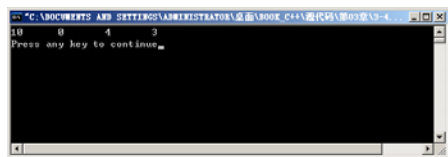


图 3-4 算术表达式

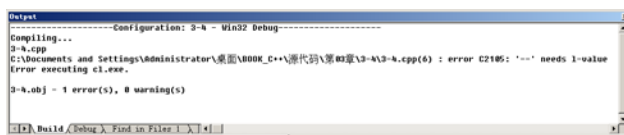


图 3-5 错误信息

3.2.2 关系表达式

由关系运算符组成的表达式为关系表达式。关系表达式的运算结果为逻辑型,常用在条件语句和循环语句中。同样,此处也通过一个简单程序段查看关系表达式的使用。

【范例 3-5】关系表达式的应用。如代码清单 3-5 给出了多个关系表达式,在程序中计算这些表达式的结果并输出,读者试分析其最终执行结果。



代码清单 3-5

```

1  #include<iostream.h>
2  void main()
3  {
4      char x='m',y='n';           //定义字符型变量并赋值
5      int n;
6      n=x<y;                       //n 的值为 true
7      cout<<n<<endl;
8      n=x==y-1;                   //n 的值为 true
9      cout<<n<<endl;
10     n=('y'!='Y')+(5<3)+(y-x==1); //n 的值为 2
11     cout<<n<<endl;
12 }

```

【运行结果】读者知道，在关系运算中，关系运算的结果为真时值等于 1，结果为假时值等于 0。那么上述程序段在 Visual C++中执行的结果如图 3-6 所示。

【范例解析】范例 3-5 代码中，由于字符之间的比较是采用 ASCII 码的比较，而字母“m”的 ASCII 码值要小于“n”的值，因此表达式 $x < y$ 成立，因此第一个 n 的值为 true，即为 1。字母“m”的 ASCII 码值比“n”的 ASCII 值要小 1，因此表达式 $x == y - 1$ 是成立的，因此第二个 n 的值也为 1。第三个表达式 $(y != 'Y') + (5 < 3) + (y - x == 1)$ 中，字母“y”与“Y”的 ASCII 值不同，小写字母的 ASCII 码要大于大写字母，因此 $(y != 'Y')$ 的值为 1，而 $(5 < 3)$ 的值为 0，同时 $(y - x == 1)$ 的值为 1，因此第三个 n 的值为 $1 + 0 + 1$ ，即结果为 2，因此就得到了如图 3-6 所示的结果。



图 3-6 关系表达式



注意 在字符的比较中，是比较字符的 ASCII 码大小。其中，字符往后，ASCII 码越大，而小写字母的 ASCII 码大于大写字母的 ASCII 码。

3.2.3 逻辑表达式

由逻辑运算符组成的表达式称为逻辑表达式。逻辑表达式的值为逻辑型，其结果也为 true 和 false，但程序中则表示为 1 和 0。

在由&&和||运算符组成的逻辑表达式中，C++规定：只对能够确定整个表达式值所需要的最少数目的子表达式进行计算。也就是说，当计算出一个子表达式之后便可确定整个逻辑表达式的值时，后面的子表达式就不需要再计算了，整个表达式的值就是该子表达式的值，这种表达式也称为短路表达式。

【范例 3-6】逻辑表达式的应用。该范例计算一个逻辑表达式的结果，并将其结果输出，实现代码如代码清单 3-6 所示。

代码清单 3-6

```

1  #include<iostream.h>
2  void main()
3  {
4      int a=3,b=0;                //声明两个整型变量并赋初值
5      int result;                 //声明一个存放结果的整型变量
6      result=!a&&a+b&&a++;        //result 的值为 0
7      cout<<result<<endl;        //输出结果
8      result=!a||a++||b++;       //result 的值为 1

```

```

9      cout<<result<<endl;
10 }

```

【运行结果】上述程序中使用了两个逻辑表达式，分别计算其值并输出，在 Visual C++ 中编译后，其执行结果如图 3-7 所示。

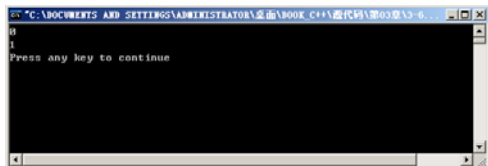


图 3-7 逻辑表达式

【范例解析】范例 3-6 程序中，第一个逻辑表达式 `!a&&a+b&&a++` 是一个由 `&&` 组成的逻辑表达式，从左至右计算三个子表达式，只要有一个为 0 就不再计算其他子表达式。当计算 `!a` 的值为 0 时，便可确定整个表达式的值为 0，因此后面的子表达式就不再计算了。因此，`result` 的值为 0。

第二个逻辑表达式 `!a||a++||b++` 是一个由 `||` 组成的逻辑表达式，从左至右计算三个子表达式，只要有一个结果为真则不再计算后面的子表达式。第一个子表达式为 `!a` 结果为 0，再计算 `a++` 结果为 4，所以就不再计算后面的子表达式。因此，`result` 的值为 0。



提示 短路表达式只需要计算许多子表达式中的若干个，这样可以大大减小计算机的运算量。

3.2.4 条件表达式

由三目运算符“?:”组成的表达式为条件表达式。例如，`a>b?x=4:x=9`; 条件表达式的值取决于?前面的表达式的值，该表达式的值为非 0 时，整个表达式的值为“:”符前面的表达式的值，否则为“:”符后面的表达式的值。

【范例 3-7】条件表达式的应用。该范例使用到了两个条件表达式，并将其结果赋值给变量，输出该变量的结果，实现代码如代码清单 3-7 所示。

代码清单 3-7

```

1  #include<iostream.h>
2  void main()
3  {
4      int a=3,b=4,c;                //声明 3 个整型变量，两个赋初值
5      c=a>b?++a:++b;                //c 的值为 5
6      cout<<a<<" "<<b<<" "<<c<<endl;
7      c=a-b?a-3?b:b-a:a;            //c 的值为 2
8      cout<<a<<" "<<b<<" "<<c<<endl;
9  }

```

【运行结果】上述程序中使用了两个条件表达式，分别计算其值并输出，在 Visual C++ 中编译后，其执行结果如图 3-8 所示。

【范例解析】范例 3-7 程序中，第一个条件表达式 `a>b?++a:++b` 中，表达式 `a>b` 为假，那么整个表达式的值取 `++b` 的值，也即变量 `c` 的值为 5，而 `b` 经过 `++b` 的运算其值也为 5，因此第一个输出 `a`、`b`、`c` 分别对应的值为 3、5、5。

第二个条件表达式 `a-b?a-3?b:b-a:a` 包含潜嵌套的条件表达式，根据条件表达式的结合性，先计算后一个条件表达式的值，表达式 `a-3` 的值为 0，则取 `b-a` 的值 2，也即子条件表达式的值为 2，而表达式 `a-b` 的值为非 0，那么整个表达式的值取子条件表达式的值 2，因此第二个输出 `a`、`b`、`c` 分别对应的值为 3、5、2。

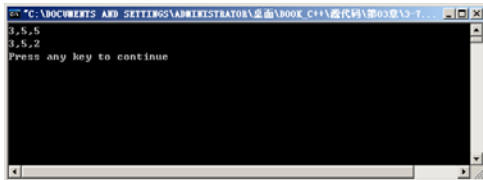


图 3-8 条件表达式



在条件表达式中还可以嵌套条件表达式，此时要先计算最里层的条件表达式的值，将其值替代表达式后依次往外计算，直到完成最外层条件表达式的计算，得到结果为止。

3.2.5 赋值表达式

由赋值运算符组成的表达式为赋值表达式。赋值运算符除了“=”之外还有 10 个复合运算符，这是赋值和运算相结合的运算符。赋值运算符的结合性是由右至左，因此，C++ 程序中允许出现连赋值的情况。例如，下面的赋值是合法的。

```
int a,b,c,d;
a=b=c=d=5/2;
```

此处先计算 $5/2$ 结果为 2，再赋值给 d，结果 $d=5/2$ 表达式的值为 2，再将这个值赋给 c，依此类推，结果 a、b、c、d 的值均为 2。

在计算复合赋值运算符表达式中，首先计算右值表达式的值后再与左值运算。例如：

```
int a=3, b=4;
a*=b+1;
```

此处先计算 $b+1$ 等于 5，再与 a 相乘赋值给 a，结果等于 15。

【范例 3-8】赋值表达式的应用。该范例包含多个赋值表达式，在 Visual C++ 中运行，读者可查看其执行效果是否与预计的相同，实现代码如代码清单 3-8 所示。

代码清单 3-8

```
1  #include<iostream.h>
2  void main()
3  {
4      int a,b,c,d;                                //声明 4 个整型变量
5      int m=3,n=4;
6      a=b=c=d=5/2;                                //赋值表达式
7      cout<<a<<"\t"<<b<<"\t"<<c<<"\t"<<d<<endl;
8      m*=n+1;                                       //复合赋值表达式
9      cout<<m<<"\t"<<n<<endl;
10 }
```

【运行结果】将范例 3-8 的代码段与前面的两个示例结合起来，并放在 Visual C++ 中执行，其执行结果如图 3-9 所示。读者可以看到其运行结果与预期的相同。

【范例解析】范例 3-8 代码中，包含了一个对多个变量同时赋值的赋值表达式 $a=b=c=d=5/2$ ，其运算顺序为从右到左，最终结果是变量 a、b、c 和 d 的值都为 $5/2$ 整除运算的结果 2。使用复合运算符“*”后，得出 m 的值并将其输出。

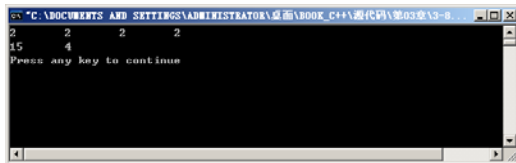


图 3-9 赋值表达式



上述代码第 7 行中输出变量值的语句中，“\t”表示两个输出值之间相隔一个 TAB 字符的距离，以便区分。

3.2.6 逗号表达式

逗号表达式是用逗号将若干个表达式连起来组成的表达式，其值是组成逗号表达式的若干个子表达式中的最后一个子表达式的值，类型也是最后一个子表达式的类型。

【范例 3-9】逗号表达式的应用。该范例包含逗号表达式，将结果赋值给变量后将其输出，实现代码如代码清单 3-9 所示。

代码清单 3-9

```

1  #include<iostream.h>
2  void main()
3  {
4      int a,b,c;                //声明 3 个整型变量
5      a=1,b=2,c=a+b+3;         //赋值
6      cout<<a<<','<<b<<','<<c<<endl;
7      c=(a++,a+=b,a-b);        //逗号表达式
8      cout<<a<<','<<b<<','<<c<<endl;
9  }
```

【运行结果】在 Visual C++ 中执行上述代码后的输出结果如图 3-10 所示。

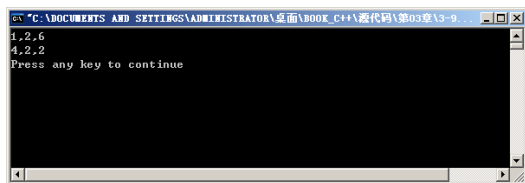


图 3-10 逗号表达式

【范例解析】范例 3-9 代码中，第一次输出的结果为 a、b、c 对应的值，根据赋值语句，其值分别为 1，2，6。第二次输出是经过逗号表达式，整个表达式的值为最后一个表达式 a-b 的值，也即 a++ 后 a 为 2，a+=b 后 a 为 4，a-b 后 a 为 2，那么整个表达式的值为 2，即 c 的值为 2。因此，经过该表达式后，a、b、c

对应的值为 4，2，2。



提示 逗号表达式在实际程序中应用不是太多，一般只在特定的场合中才需要使用。

3.3 语句

3.2 节介绍了表达式，读者都知道表达式是由运算符和常量、变量构成的，而本节要介绍的语句就是由表达式来构成的。一般说来，C++ 语句和表达式并没有严格区分。一个表达式，加上一个分号后，便直接形成语句。例如，算术表达式 3+2，为其加上分号，写成如下形式：

```
3 + 2;
```

这就是语句了。计算机可以执行该语句，但它并不改变程序的运行逻辑。当然，这条语句并没有实际意义。当一些表达式组合起来，完成某一相对完整的功能后，再加一个分号表示结束，这就组成一条语句。如下面的语句：

```
a = 3 + 2;
```

这就是一条赋值语句，它改变了 a 的值。

3.3.1 语句中的空格

在前面的示例中，读者接触到了许多简单的程序，其中包含很多语句。读者可能也注意到了，程序段中的语句并不是全部顶格对齐的，在语句中包含了很多空格。

【范例 3-10】语句中存在部分空格，保证程序的可读性。该范例是一个简单地比较两个变量大小的程序，输出其中大的一个，实现代码如代码清单 3-10 所示。



代码清单 3-10

```

1  #include <iostream.h>
2  void main()
3  {
4      int a,b;
5      cin>>a;                //接收输入
6      cin>>b;
7      if (a>b)                //比较大小, 如果 a>b 成立
8          cout<<"the max number is: "<<a<<endl;    //输出较大值
9      else                    //如果 a<b
10         cout<<"the max number is: "<<b<<endl;
11 }

```

【运行结果】在 Visual C++ 中, 编译器系统将自动给某些语句留出一定的空格, 读者在编辑区中输入代码即可发现。运行上述程序, 其结果如图 3-11 所示。

【范例解析】从范例 3-10 代码中, 读者可以看出, 其语句之间存在着许多的空格, 每条语句并不都是对齐的, 而是参差不齐的。这在程序中是非常必要的, 这样才能使得程序的结构更加清晰, 从而提高程序的可读性。如果所有语句都是排列整齐的, 那么会给阅读该程序的用户造成很大困难。

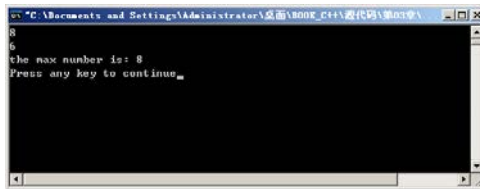


图 3-11 比较大小

此外, 在一条语句之间也存在空格, 如 if (a>b) 语句, if 关键字和 (a>b) 表达式之间含有空格, 这都是为了提高程序的可读性而做的。



注意 在语句之间和语句内含有一定的空格, 是养成良好的编程风格的重要方面, 这样的程序可提高程序的可读性。

3.3.2 空语句

前面使用较多的都是表达式语句, 即一个表达式加上一个分号组成。除此之外, 语句也可以直接是一个分号, 这种语句称为空语句。空语句仅由一个分号组成, 不进行任何操作。一般用于语法上要求有一条语句但实际没有任何操作的场合。例如下列语句中:

```

for(i=1;i<10;i++)
;                //空语句, 起延时作用

```

for 循环中没有进行任何操作, 而只是起到延时的作用。一般情况下, 除非为了调试程序方便, 否则写一句空语句也纯属多余。

3.3.3 声明语句

变量的声明语句在前面程序段中已经使用很多了, 其主要作用是完成指定变量的定义。声明语句的基本格式如下:

```
<数据类型> <变量 1>...<变量 n>;
```

在用声明语句声明变量时, 既可以一条语句只声明一个变量, 也可以在一条语句中声明多个变量, 例如:

```

int a;
int b,c,d;

```




上面这两种形式的声明语句都是允许的。然而需要读者注意的是，当在一条语句中声明多个变量时，声明变量的数据类型必须一致，也就是说，不允许在一条语句中声明多个数据类型不一样的变量。

此外，在声明变量的同时，可以和赋值语句结合起来为变量赋初值，这在前面范例中也使用较多。例如：

```
int a=4;
int b=1,c,d=2;
```

这些写法在声明语句中都是允许的，这是为了在定义变量时同时对其初始化。

3.3.4 赋值语句

与声明语句不同，赋值语句实现为指定变量获得指定值的操作。例如：

```
a = 20;
b = 10 * 2 / 3;
c = 2 * (a + b);
```

如上程序代码所示，使用赋值语句给变量赋值时，右值（等号右边的值），可以是一个简单的常数或变量，也可以是一个表达式。

在 C++ 中，赋值语句可以使用连等，例如：

```
a = b = 10;
```

执行这一语句时，先将 10 赋值给变量 b，然后将 b 中的值赋给 a，结果 a 和 b 的值都是 10。此外，赋值语句右边还可以是赋值语句。

本节介绍了几种常用的语句，在后续章节中还将介绍 C++ 的流程控制语句，C++ 程序就是由一行行语句组成的。

3.4 小结

本章主要介绍了 C++ 的运算符和表达式，这是 C++ 的入门基础之一。在运算符部分，本章着重讲解了 7 种运算符：算术运算符、赋值运算符、关系运算符、逻辑运算符、条件运算符、逗号运算符、位运算符，在介绍这些运算符的基本概念后，都设置了一个简单示例用以说明这些运算符在具体程序中的使用，并对这些运算符的优先级和结合性进行了说明。同样，针对不同的运算符，在表达式部分也对这些表达式做了详细介绍。最后简要介绍了 C++ 中常见的几种语句，在第 4 章中还将继续讲解 C++ 的流程控制语句。

3.5 习题

1. 编写一个 C++ 程序，要求从键盘上输入两个整数，将其存入整型变量 x 和 y，并求出这两个整数进行四则算术运算、整除运算和取余运算的结果。例如，在用户屏幕输入 2 和 5 后，其返回结果如图 3-12 所示。

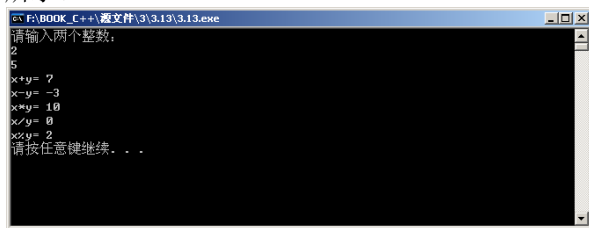


图 3-12 算术运算



【解答】该习题首先必须声明两个整型变量，并将用户从键盘输入的值存入两个变量，对其进行算术运算并输出结果，可调用本章介绍的对应算术运算符，将结果存入到一个整型变量后输出。需要读者注意的是，取余运算的两个操作数必须为整数。其实现的简要代码如下所示。

```
int x,y,z;
cout<<"请输入两个整数: "<<endl;
cin>>x>>y;
z=x+y;
cout<<"x+y= "<<z<<endl;
z=x-y;
cout<<"x-y= "<<z<<endl;
z=x*y;
cout<<"x*y= "<<z<<endl;
z=x/y;
cout<<"x/y= "<<z<<endl;
z=x%y;
cout<<"x%y= "<<z<<endl;
```

2. 若有下面的语句:

```
int x=5,y=6,z=7,m;
```

则在计算表达式:

```
m=(x<z-4)
```

后, m 的值为多少?

【解答】该习题主要考查关系运算符，此外读者还需要考虑算术运算符与关系运算符的优先级问题。对于表达式 $x < z - 4$ 而言，先运算算术表达式 $z - 4$ ，其值为 3，再运算关系表达式 $x < 0$ ，此时 x 的值为 5，因此表达式 $x < z - 4$ 不成立，其值为假，因此 m 的值为 0。

3. 分析下面程序的输出结果。

```
#include <iostream.h> //包含头文件
void main()
{
    int x=10,result; //定义整型变量
    double y=8.5; //定义双精度变量
    result=x++; //递增表达式
    cout<<"result= "<<result<<"\t"<<"x= "<<x<<endl; //输出提示
    result=--x; //递减表达式
    cout<<"result= "<<result<<"\t"<<"x= "<<x<<endl;
    result=x>y; //关系表达式
    cout<<"result= "<<result<<endl;
    result=x>0&&y<0; //逻辑表达式
    cout<<"result= "<<result<<endl;
    result=!x || y<1; //逻辑表达式
    cout<<"result= "<<result<<endl;
    result=(x++,y+=x,x-y); //逗号表达式
    cout<<"result= "<<result<<"\t"<<"x= "<<x<<"\t"<<"y= "<<y<<endl;
    result=(x>y?x++:y); //条件表达式
    cout<<"result= "<<result<<"\t"<<"x= "<<x<<endl;
}
```

【解答】该程序中，给出了多个表达式在具体使用环境中值的变化，尤其是递增和递减表达式，其将变量运算的结果赋值给表达式后，变量自身会改变。而关系表达式和逻辑表达式的结果只有两种：0 和 1。当表达式的结果为真时返回 1，反之返回 0。逗号表达式的结果为表达式中最后一个表达式的值，条件表达式的结果依赖于表达式中的关系表达式。其输出结果如图 3-13 所示。

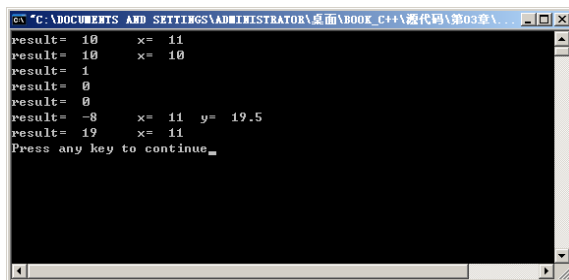


图 3-13 表达式的应用

4. 设 `int x = 15`, 则表达式 `x <= 20 ? 10 : 30` 的值为多少?

【解答】该习题主要考查条件运算符, 而条件运算符为一个三目运算符, 对于其中的三个表达式的含义为: 如果子表达式 1 的结果是 `true`, 则整个表达式的结果是子表达式 2 的值; 否则是子表达式 3 的值。因此, 该题首先判断表达式 1 即 `x <= 10` 是否成立, 此处 `x` 的值为 10, 因此该表达式不成立, 其值为表达式 2 的值, 即 30。

5. 编写一个 C++ 程序, 要求从键盘上输入两个整数, 将其存入整型变量 `a` 和 `b`, 不用第三个变量, 将变量 `a` 和 `b` 的值进行互换, 并将交换前后的 `a` 和 `b` 的值输出。例如, 输入 `a` 的值为 2, `b` 的值为 5, 输出结果如图 3-14 所示。

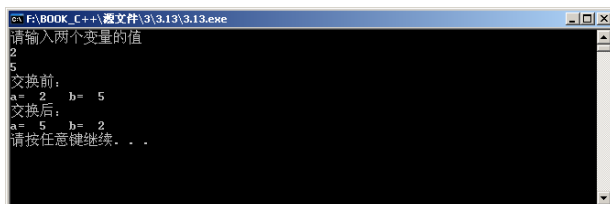


图 3-14 交换变量值

【分析】该习题主要考查递增递减运算符的应用, 不允许使用第三个变量, 要求将变量的值互换, 可以通过变量 `a` 和变量 `b` 的相互赋值来实现。比如将 `a` 的值变为 `a+b`, `b` 的值变为 `a-b`, 此时 `a` 的值为 `a+b`, 因此 `b` 的值就为 `a+b-b`, 即相当于 `b=a` 了, 此时的 `a` 为未改变前的变量 `a` 的值。依此类推, 可得出 `a=a-b`, 即可得到未改变前的变量的 `b` 的值。其简要代码如下所示:

```
int a,b;
cout<<"请输入两个变量的值"<<endl;
cin>>a>>b;
cout<<"交换前: "<<endl;
cout<<"a= "<<a<<"    "<<"b= "<<b<<endl;
a+=b;
b=a-b;
a-=b;
cout<<"交换后: "<<endl;
cout<<"a= "<<a<<"    "<<"b= "<<b<<endl;
```

6. 若表达式 `(a + b) > c * 2 && b != 5 || !(1 / 2)` 中, `a`、`b`、`c` 的定义和赋值为:

```
int a = 3, b = 4, c = 2;
```

则表达式的值为多少?

【分析】该习题涉及关系运算符、算术运算符、逻辑运算符等, 在表达式的运算前, 读者要清楚其优先级和结合性。其中算术运算符的优先级最高, 其次为关系运算符, 最后为逻辑运算符, 但其中非运算 (!) 的优先级又高于算术运算符。因此, 该表达式先计算括号, `a+b` 的值为 7, `c*2` 的值为 4, `1/2` 的值为 0。因此, 整个表达式的值为 1。



7. 下面程序段的输出结果是多少？

```
#include <iostream.h>
void main()
{
    int a,b;
    a=b=10;
    cout<<"a="<<a<<" "<<"b="<<b<<endl;
    int c=10,d;
    d=(c=5 * (a + b));
    cout<<"c="<<c<<" "<<"d="<<d<<endl;
}
```

【解答】该程序段分别输出变量 a、b、c 和 d 的值，其值 a 和 b 的值在初始化时已确定，其输出都为 10，而变量 c 和 d 的值都相同，其值都为 $5 * (a+b)$ ，即 100。因此，该程序的输出为 a=10, b=10, c=100, d=100。

第 4 章 程序控制结构

程序控制结构即结构化程序设计用到的结构，或者称为流程控制结构。和传统的程序设计语言一样，C++也具有结构化程序设计的 4 种结构：

- 顺序结构。
- 选择结构。
- 循环结构。
- 转向语句。

本章将重点介绍这几种结构在 C++中的流程控制语句及其实现。以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的。读者在学习本章内容时可以作为一个学习的参照。

- 了解 C++的面向过程的结构化设计方法。
- 熟练掌握 C++支持的三种程序结构：顺序结构、选择结构和循环结构。
- 掌握转向语句的功能及其使用。

4.1 顺序结构

顺序结构是指按照所有语句出现的顺序先后执行，先出现的先执行，后出现的后执行。顺序结构的执行流程如图 4-1 所示。

C++中，顺序结构的语句一般包含三种：表达式语句、输入语句和输出语句。

4.1.1 表达式语句

通过前面章节的学习，读者对表达式语句已经有了一定认识，在本节中再将其强化一下。表达式语句是最简单的语句，任何一个表达式加上分号就是一个表达式语句。表达式语句与表达式的区别是，表达式可包含在其他表达式中，而表达式语句则不能。表达式语句的一般形式为：

表达式；

表达式语句的特殊形式为空语句，这在第 3 章也介绍过。空语句就是只有一个分号的语句，相当于一个空表达式，其一般形式为：

；

此外，我们也应了解一下复合语句。复合语句由两条以上的语句组成，并用一对花括号({})括起来。复合语句又称为块语句或块程序，其一般形式为：

```
{  
    <语句 1>  
    <语句 2>  
    ⋮  
    <语句 n>  
}
```



图 4-1 顺序结构



空语句通常没有实际作用，主要用于延时，与循环语句一起使用实现程序的等待、延时功能。

4.1.2 输入语句

事实上，C++没有提供输入/输出语句，其输入/输出功能由函数（scanf、printf）或流控制来实现。输入/输出流（I/O 流）是输入或输出的一系列字节。C++定义了包含重载运算符“<<”和“>>”的 iostream 类。在这里只介绍如何利用 C++ 的标准输入/输出流实现数据的输入/输出功能。

基本输入/输出语句较为复杂，在前面的示例中，用到了 cin 和 cout 分别实现从键盘输入和在显示器上输出的功能。绝大多数 C++ 程序都使用了系统提供的 I/O 流，以实现基本的输入和输出操作。在 I/O 流类的定义中，把 C++ 语言中的左、右移位运算符<<和>>通过运算符重载的方法定义为插入（输出）和提取（输入）运算符。

当程序需要执行键盘输入时，可以使用抽取操作符“>>”从输入流 cin 中抽取键盘输入的字符和数字，并把它赋给指定的变量。

【范例 4-1】输入语句的使用。该范例给出了最简单的输入语句的实现。该程序段的功能是接收用户从键盘输入的一个整数，并将其存储到变量 a 中，实现代码如代码清单 4-1 所示。

代码清单 4-1

```
1  #include<iostream.h>           //包含头文件
2  void main()
3  {
4      int a;                     //定义整型变量
5      cin>>a;                    //输入语句
6  }
```

【运行结果】在 Visual C++ 中新建一个【C++ Source File】后输入如上的代码，编译运行后，其结果如图 4-2 所示。



图 4-2 输入语句

【范例解析】范例 4-1 代码中只有一个输入语句，即该程序执行后接收键盘输入的数值，并将其存储到变量 a 中，但并未对该变量进行其他任何操作。此外，cin 语句可同时接收多个键盘输入并将其存入对应的变量中，可以分辨不同的变量

类型。例如，下列的写法也是正确的。

```
int a;
double b;
cin>>a>>b;           //cin 可分辨不同的变量类型
```



此处的抽取操作符“>>”与位移运算符“>>”是同样的符号，但这个符号在不同的地方其含义是不一样的。

4.1.3 输出语句

同样，当程序需要在屏幕上显示输出时，可以使用插入操作符“<<”向输出流 cout 中插入字符和数字，并把它在屏幕上显示输出。

【范例 4-2】输出语句的使用。该范例给出了最简单的输出语句的实现，其功能是输出字符串“Hello”，使其显示到屏幕上，实现代码如代码清单 4-2 所示。

代码清单 4-2

```

1  #include<iostream.h>                                //包含输入/输出的头文件
2  void main()
3  {
4      cout<<"Hello.\n";                                //输出语句
5  }

```

【运行结果】同样，在 Visual C++ 中新建一个【C++ Source File】文件后输入如上的代码，编译运行后，其结果如图 4-3 所示。

【范例解析】范例 4-2 程序实现在终端输出一个字符串。需要注意的是，此处的字符“\n”为转义字符，表示换行。在 cout 操作符后加上该字符，表示输出后光标往下移动一行。此外，在 C++ 中，换行也可以通过关键字 endl 来实现，因此，上述语句也可以改写为：

```
cout<<"Hello. "<<endl;
```

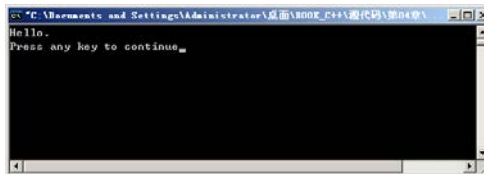


图 4-3 输出语句



注意 与输入语句一样，此处的插入操作符“<<”与位移运算符“<<”是同样的符号，但这两种符号在不同的地方其含义是不一样的。

在 C++ 程序中，cin 与 cout 允许将任何基本数据类型的名词或值传给流。而且书写格式较灵活，可以在同一行中串连书写，也可以分写在几行，提高可读性。例如，下列语句：

```
cout<<"hello";
cout<<3;
cout<<endl;
```

等价于

```
cout<<"hello"<<3<<endl;
```

也等价于

```
cout<<"hello"                                //注意：行末无分号
<<3                                           //行末无分号
<<endl;
```

4.1.4 格式控制符

前面提到过，cin 和 cout 是预先定义的流对象，分别为标准输入流和标准输出流，一般代表标准输入设备（键盘）和标准输出设备（显示器）。为了更好地调整输入/输出格式，C++ 还提供了格式控制函数和格式控制符，主要有如下 5 种。

- **flags**: 该函数一般有以下两种形式，第一种形式是通过参数 lFlags 重新设置标志字并返回原来的标志字；另一种形式是通过无参数的 flags 函数返回当前的标志字。

```
long flags(long lFlags);
long flags();
```

- **setf**: 该函数一般也有以下两种形式，第一种形式是通过参数 lFlags 来设置指定的格式控制标志位；第二种形式是用来设置指定的格式控制标志位的值。

```
long setf (long lFlags);
long setf (long lFlags,long lMask);
```

- **unsetf**: 该函数的一般形式如下，这种形式是通过参数 lFlags 来清除指定的格式控制标志位（使那些位的值为“0”）。



```
long unsetf (long lFlags);
```

- **fill**: 该函数一般有以下两种形式, 第一种形式是将填充字符设置为 `cFill`, 并返回原填充字符; 第二种形式是通过无参数的 `fill` 函数返回当前的填充字符。

```
char fill (char cFill);
char fill ();
```

- **precision**: 该函数形式一般也有以下两种, 第一种形式是设置浮点数精度为 `np`, 并返回原精度; 第二种形式是通过无参数的 `precision` 函数返回当前的浮点数精度。

```
int precision (int np);
int precision ();
```

- **width**: 该函数的形式一般也有以下两种, 第一种形式设置当前显示数据的域宽 `nw`, 并返回原域宽; 第二种形式是通过无参数的 `width` 函数返回当前显示数据的域宽。

```
int width (int nw);
int width ();
```

C++在头文件 `iomanip.h` 中定义了控制符对象, 可以直接将这些控制符嵌入到 I/O 语句中进行格式控制。在使用这些控制符时, 只需在程序的开头包含头文件 `iomanip.h`, 即使用 `#include<iomanip.h>` 预处理命令即可。



警告 使用 `cin` 和 `cout` 必须加入头文件 `<iostream.h>`, 即在程序代码的开始部分使用预处理命令 `#include<iostream.h>`, 否则将不能通过系统编译。

为了让读者更好地理解使用格式控制函数和格式控制符显示不同输出格式的数据, 下面通过一个示例来讲解。

1. 控制不同进制的输出

前面讲解了众多格式控制符, 为方便读者理解, 这里将重要的控制符单独列出, 通过示例的形式介绍, 使读者能更深刻地理解。

【范例 4-3】控制不同进制的输出。该范例通过前面介绍的 I/O 控制符来控制不同进制的数值输出, 实现代码如代码清单 4-3 所示。

代码清单 4-3

```
1  #include<iostream.h>
2  void main()
3  {
4      int a=1000;
5      cout<<"默认下: "<<a<<endl;           //默认输出进制
6      cout<<"十进制: "<<dec<<a<<endl;       //输出十进制
7      cout<<"八进制: "<<oct<<a<<endl;       //输出八进制
8      cout<<"十六进制: "<<hex<<a<<endl;     //输出十六进制
9  }
```

【运行结果】在 Visual C++中编译上述程序代码, 若编译无误后使用快捷键 **【Ctrl+F5】** 执行该代码, 其返回结果如图 4-4 所示。

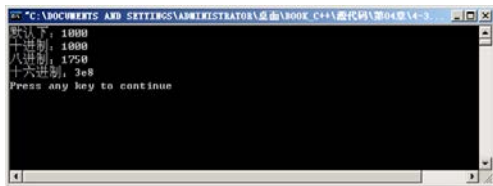


图 4-4 控制不同进制的输出

【范例解析】范例 4-3 中, 输出整数 1000 在各种不同进制下的表示, 默认输出是十进制的数字, 不需要任何格式输出符即为默认输出格式。



格式输出符 `dec` 也表示十进制输出, 通常省略。格式输出符 `oct` 表示为八进制输出, `hex` 表示十六进制输出。

2. 控制输出宽度

在具体的应用中, 通常需要对输出内容的宽度做一些限制, 比如设置为输出若干位, 而这可以通过 4.1.4 节讲解的控制符来实现。

【范例 4-4】控制输出宽度。该范例针对同一个数值, 通过输出格式控制函数, 实现不同宽度的输出格式, 实现代码如代码清单 4-4 所示。

代码清单 4-4

```
1  #include<iostream.h>
2  #include<iomanip.h>           //包含输出格式的头文件
3  void main()
4  {
5      int a=1234567890;         //定义变量
6      double b=123.45;
7      cout<<setw(10)<<a<<endl;   //输出宽度为 10
8      cout<<setw(10)<<b<<endl;
9      cout<<setw(8)<<b<<endl;     //输出宽度为 8
10     cout<<setw(6)<<b<<endl;     //输出宽度为 6
11     cout<<setw(4)<<b<<endl;     //输出宽度为 4
12 }
```

【运行结果】在 Visual C++ 中编译上述程序代码, 若编译无误后使用快捷键 **【Ctrl+F5】** 执行该代码, 其返回结果如图 4-5 所示。

【范例解析】范例 4-4 实现对输出字符或数值宽度的控制。读者从上述执行结果可以看出, 当需输出的内容宽度达不到 `setw` 控制符中设定的宽度时, Visual C++ 将在输出内容前面填补空格; 当需输出的内容宽度大于 `setw` 控制符中设定的宽度时, 系统将全部予以输出。

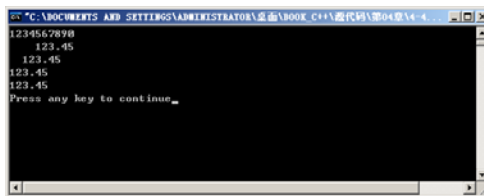


图 4-5 控制输出宽度



上述输出语句中的 `endl` 控制符表示在输出时插入换行符并刷新流, 其也可以用 C 语言中的 “`\n`” 来代替。

3. 控制输出精度

在实际应用中, 有时需要对输出的浮点型数据进行精度控制, 这也需要通过输出控制符来实现。

【范例 4-5】控制输出精度。该范例针对同一个数值, 通过输出控制符, 实现不同精度的输出, 实现代码如代码清单 4-5 所示。

代码清单 4-5

```
1  #include<iostream.h>
2  #include<iomanip.h>
3  void main()
4  {
5      double a=1.23456789;
6      cout<<a<<endl;           //输出默认格式的变量值
```




```
7      cout<<setprecision(3)<<a<<endl;           //输出精度为 3
8      cout<<setprecision(10)<<a<<endl;         //输出精度为 10
9  }
```

【运行结果】同样，在 Visual C++中编译上述程序代码，若编译无误后使用快捷键【Ctrl+F5】执行该代码，其返回结果如图 4-6 所示。

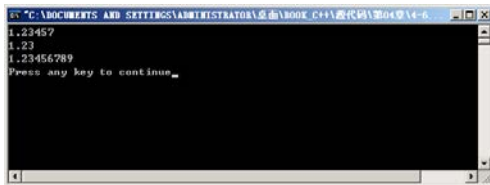


图 4-6 控制输出精度

【范例解析】范例 4-3 对一个浮点数控制其输出的有效位数。从上述示例中，读者可以看出，如果希望显示的数字是 1.23，即保留两位小数，可用 setprecision（3）控制符加以控制，此时显示 3 位有效位。当小数位数截短显示时，进行四舍五入处理。此外，需要注意的是，C++默认的输出流数值的有效位是 6。

注意 凡是在程序中使用了格式控制符，诸如以上的控制进制输出、控制输出的宽度、精度等，在预编译时都必须包含头文件 iomanip.h，即在头文件处加上语句#include<iomanip.h>，否则编译时将不能通过。

此外，C++还提供了其余一些控制符，表 4-1 所示为常用的 I/O 流控制符。

表 4-1 常用的 I/O 流控制符

控制符	描述
dec	置基数为 10
hex	置基数为 16
oct	置基数为 8
setfill(c)	设填充字符为 c
setprecision(n)	设显示小数精度为 n 位
setw(n)	设域宽为 n 个字符
setiosflags(ios::fixed)	固定的浮点显示
setiosflags(ios::scientific)	指数表示
setiosflags(ios::left)	左对齐
setiosflags(ios::right)	右对齐
setiosflags(ios::skipws)	忽略前导空白
setiosflags(ios::uppercase)	十六进制数大写输出
setiosflags(ios::lowercase)	十六进制数小写输出

4.1.5 应用示例

本节介绍了顺序结构的组成语句，主要包括表达式语句、输入语句和输出语句。下面通过一个较为综合的示例回顾一下顺序结构的各种语句及控制函数和控制符。

【范例 4-6】顺序结构应用示例。该范例给出了使用不同格式控制函数和格式控制符输出数据时返回的数据显示，实现代码如代码清单 4-6 所示。

代码清单 4-6

```
1  #include <iostream.h>           //预处理文件
2  #include <iomanip.h>
```

```

3  void main()                                //主函数
4  {
5      const double Num=123.4567;
6      cout.flags (ios::right);                //设置对齐的标志位是右
7      cout<<setw(10)<<Num<<endl;              //显示数据的域宽是 10
8      cout.fill ('*');                        //填充字符 '*'
9      cout.width (12);                        //显示数据的域宽是 12
10     cout<<Num<<endl;
11     cout.precision (4);                     //浮点数的有效个数为 4
12     cout<<Num<<endl;
13     cout.setf(ios::showpos);                //显示正号
14     cout.precision (2);                     //浮点数的有效个数为 4
15     cout<<Num<<endl;
16     cout.unsetf(ios::showpos);              //显示正号
17     int n;
18     cout<<"输入一个八进制整数: ";
19     cin>>oct>>n;                            //输入一个八进制数
20     cout<<"八进制数 n 是: " <<oct<<n<<endl;
21     cout<<"对应的十进制数是: " <<dec<<n<<endl;
22     cout<<"对应的十六进制数是: " <<hex<<n<<endl; //输出
23 }

```

【运行结果】在 Visual C++ 6.0 集成环境下执行上述代码，其执行结果如图 4-7 所示。

【范例解析】范例 4-6 程序中，设置对齐的标志位是向右，其输出的宽度为 10，如不够则在前面填充*符号，输出精度为 4，显示数值的正号。此外，接收一个键盘输入的八进制数值，以不同进制将该数值输出。

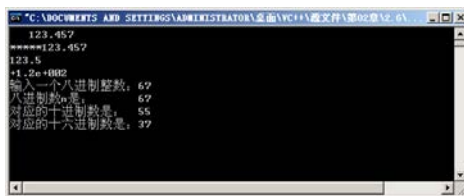


图 4-7 应用示例



提示 上述代码中使用了 4.1 节中所提到的所有格式控制函数和格式控制符，读者可根据其注释和如图 4-5 所示的输出结果理解这些控制函数或控制的作用。

4.2 选择结构

选择结构是用来判断所给定的语句是否满足条件，根据判断结果，选择执行不同的分支语句。常用的语句有如下 4 种语句：if 语句、if...else 语句、多重 if...else 语句和 switch 语句。下面将详细介绍。

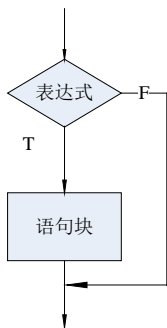


图 4-8 if 语句执行流程

4.2.1 if 语句

if 语句为单分支条件语句，其说明语句的一般形式为：

```
if (<表达式> )
    <语句>;
```

其中，表达式可以是 int 型、long 型、char 型和 enum 型，其值有 0 和非 0 两种，0 为 false，非 0 为 true。语句可以是任何类型的语句，也可以是块语句。该语句的作用是：如果表达式的值为 true，则执行 if 后面的语句；否则跳过执行后面的语句。其执行顺序如图 4-8 所示。

【范例 4-7】if 语句的应用。已知两个变量 x 和 y，比较它们的大小，当 x 小于 y 时，交



换这两个变量的值，使 x 的值大于 y。实现程序如代码清单 4-7 所示。

代码清单 4-7

```
1  #include <iostream.h> //预处理文件
2  void main()           //主函数
3  {
4      int x,y,temp;      //定义变量
5      cout<<"Please input 2 numbers:";
6      cin>>x>>y;        //接收用户输入
7      if (x<y)           //判断 x,y 的大小
8      {
9          temp=x;
10         x=y;
11         y=temp;
12     }                  //x 小于 y 则交换
13     cout<<"x="<<x<<endl;
14     cout<<"y="<<y<<endl; //输出结果
15 }
```

【运行结果】在 Visual C++中执行上述程序，其执行结果如图 4-9 所示。

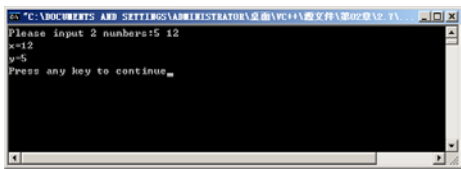


图 4-9 if 语句

【范例解析】读者可以看到，范例 4-7 程序代码中使用了 if 语句，当表达式 x<y 成立时，执行{}内的语句块，完成交换操作，使得变量 x 的值更大；当表达式 x<y 不成立时，则不执行{}内的任何操作，直接输出变量 x 和 y 的值，也使得变量 x 的值更大，这样就达到了示例的要求。

注意 if 语句一般用于有多种可能的情况，但只有一种情况需要进行操作，即其他情况不需要进行程序控制的情况下。

4.2.2 if...else 语句

if...else 语句为双分支条件语句，其说明语句的一般形式为：

```
if(<表达式>)
    <语句 1>;
else
    <语句 2>;
```

该语句的作用是：如果表达式的值为 true，则执行语句 1；否则执行语句 2。相对于如上的 if 语句，if...else 语句增加了对表达式的值为 false 时的处理语句，其执行流程如图 4-10 所示。

【范例 4-8】if...else 语句的应用。该范例根据输入的学生的百分制成绩，判断该学生是否及格，其实现程序如代码清单 4-8 所示。

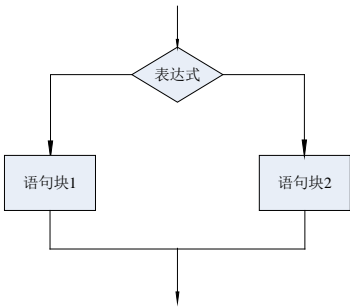


图 4-10 if...else 语句执行流程

代码清单 4-8

```
1  #include <iostream.h> //预处理文件
2  void main()           //主函数
3  {
4      int score;        //定义变量
```

```

5      cout<<"请输入学生分数: "<<endl;
6      cin>>score;                      //接收用户输入分数
7      if (score>=60)                   //分数大于或等于 60
8          cout<<"及格"<<endl;
9      else                             //分数小于 60
10         cout<<"不及格"<<endl;        //双分支判断
    }

```

【运行结果】在 Visual C++ 中执行上述程序，其执行结果如图 4-11 所示。

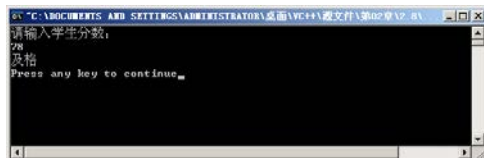


图 4-11 if...else 语句

【范例解析】范例 4-8 代码的结构非常清晰。定义接收用户输入分数的变量，给出提示信息，接收用户输入，再根据用户输入使用 if...else 语句判断是否及格。



提示 if...else 语句一般用于两种情况，其一，根据其是否满足条件而分别执行不同的程序段，其二在进行不同操作的情况下。

4.2.3 多重 if...else 语句

多重 if...else 语句为多分支条件语句或 if...else if...else 语句，其说明语句的一般形式为：

```

if (<表达式 1>)
    <语句 1>;
else if (<表达式 2>)
    <语句 2>;
...
else if (<表达式 n>)
    <语句 n>;
else
    <语句 n+1>;

```

该语句形式中，首先测试表达式 1，如果它为 false，接着就测试表达式 2，依此类推，直到找到一个为 true 的条件就执行相应的语句块，执行后即跳出该语句。如果所有的条件都不是 true，则执行 else 语句块。其执行顺序如图 4-12 所示。

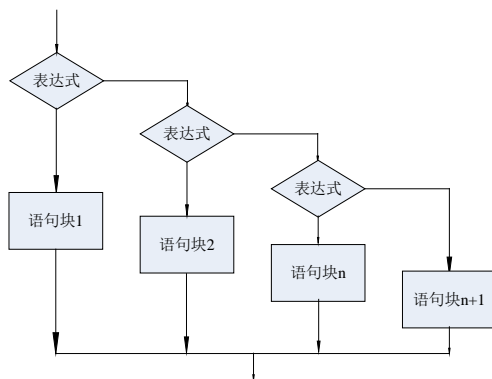


图 4-12 多重 if...else 语句



注意 该语句为 if 语句的嵌套，在嵌套时，C++ 语言规定每个 else 只与其前面最近的未配对的 if 配对，也可以用 { } 确定层次关系。

【范例 4-9】多重 if...else 语句的应用。该范例输入学生的百分制成绩，判断该学生的等级。其等级评定条件如下：

等级=	{	优	score >= 90
		良	80 <= score < 90
		中	70 <= score < 80
		及格	60 <= score < 70
		不及格	score < 60

读者可以看出该示例的等级有 5 种情况，因此可以使用多重分支语句 if...else if...else 来实现。其实现程序如代码清单 4-9 所示。

代码清单 4-9

```

1  #include <iostream.h>
2  void main()
3  {
4      int score;
5      cout<<"请输入学生分数: "<<endl;
6      cin>>score;                                //接收用户输入分数
7      if (score>=90)                               //多重 if...else 语句
8          cout<<"优"<<endl;
9      else if(score>=80)                           //分数在 80~89 之间
10         cout<<"良"<<endl;
11     else if(score>=70)                           //分数在 70~79 之间
12         cout<<"中"<<endl;
13     else if(score>=60)                           //分数在 60~69 之间
14         cout<<"及格"<<endl;
15     else                                           //分数小于 60
16         cout<<"不及格"<<endl;                   //多分支判断
17 }
```

【执行结果】范例 4-9 程序代码使用了多重分支语句 if...else if...else，根据用户输入的分进行等级的判断，其程序执行结果如图 4-13 所示。

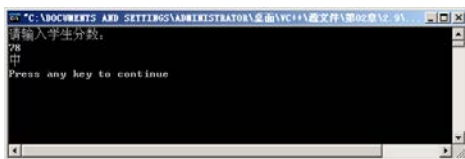


图 4-13 多重 if...else 语句

【范例解析】范例 4-9 代码中，首先接收用户从键盘输入的分，将其存入变量 score 中，再使用多重 if...else 语句判断该分数在哪一区间内，并对应输出不同的等级。

【范例 4-10】读者可以发现，其实多重分支语句 if...else if...else 可以使用 if 语句的嵌套来实现。例如，针对上述示例，可以使用如代码清单 4-10 所示的程序来实现相同功能。

代码清单 4-10

```

1  #include <iostream.h>
2  void main()
```

```

3  {
4      int score;
5      cout<<"请输入学生分数: "<<endl;
6      cin>>score;                                //接收用户输入分数
7      if (score>=60)                              //分数大于 60
8      {
9          if (score>=70)                          //分数大于 70
10         {
11             if (score>=80)                      //分数大于 80
12             {
13                 if (score>=90)                //分数大于 90
14                     cout<<"优"<<endl;
15                 else                          //分数小于 90 但大于 80
16                     cout<<"良"<<endl;
17             }
18             else                              //分数小于 80 但大于 70
19                 cout<<"中"<<endl;
20         }
21         else                                  //分数小于 70 但大于 60
22             cout<<"及格"<<endl;
23     }
24     else                                      //分数小于 60
25         cout<<"不及格"<<endl;                //多分支判断
26 }

```

上述代码中使用了多重分支嵌套，其使用多个 if...else 语句的嵌套实现多重 if...else 语句的功能。在 Visual C++ 6.0 中运行上述代码，读者可发现其执行结果与图 4-13 相同。



警告 在实际使用中，一般不推荐使用多重分支嵌套，这是因为多重分支嵌套很容易出错，对于程序的可读性也有损害。

4.2.4 switch 语句

switch 语句也称情况语句，其也是一种多分支语句，其说明语句的一般形式为：

```

switch ( <表达式> )
{
case <常量表达式 1>;
    <语句 1>;break;
case <常量表达式 2>;
    <语句 2>;break;
:
case <常量表达式 n>;
    <语句 n>;break;
default:
    <语句 n>;break;
}

```

该语句的执行顺序是：首先计算 switch 表达式后的值，然后将其结果与 case 后面的各常量表达式进行比较，若匹配，则执行该分支后的语句，执行完后，遇到 break 语句，则退出 switch 语句；若其结果与 case 后面的各常量表达式都不匹配，则执行 default 后面的语句。其执行流程如图 4-14 所示。

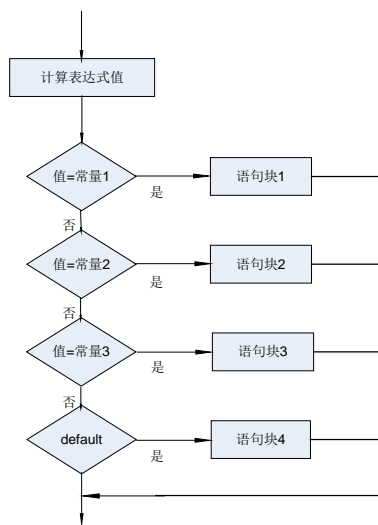


图 4-14 switch 语句执行流程



【范例 4-11】switch 语句的应用。该范例输入 0~6 的整数，转换成星期输出。该范例可以使用 switch 语句来实现，也可以使用 4.2.3 节提到的多重分支和分支嵌套的方法来实现，为简单起见，这里只给出使用 switch 语句实现的程序代码如代码清单 4-11 所示。

代码清单 4-11

```
1  #include <iostream.h>
2  void main ()
3  {
4      int day;                                //定义变量
5      cout<<"请输入 0~6 的数值: "<<endl;
6      cin>>day;                                //接收输入
7      switch(day)                             //switch 语句
8      {
9          case 0:                             //输入 day 值为 0
10             cout<<"星期日"<<endl;
11             break;
12          case 1:                             //输入 day 值为 1
13             cout<<"星期一"<<endl;
14             break;
15          case 2:                             //输入 day 值为 2
16             cout<<"星期二"<<endl;
17             break;
18          case 3:                             //输入 day 值为 3
19             cout<<"星期三"<<endl;
20             break;
21          case 4:                             //输入 day 值为 4
22             cout<<"星期四"<<endl;
23             break;
24          case 5:                             //输入 day 值为 5
25             cout<<"星期五"<<endl;
26             break;
27          case 6:                             //输入 day 值为 6
28             cout<<"星期六"<<endl;
29             break;
30          default:                            //输入 day 值为其他
31             cout<<"输入非法"<<endl;
32      }
33 }
```

【运行结果】上述程序执行结果如图 4-15 所示。

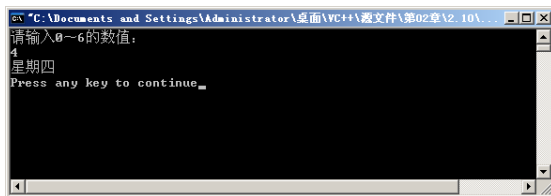


图 4-15 switch 语句

【范例解析】范例 4-11 首先从键盘接收一个 0~6 之间的整型数值，将其存入变量 day 中，再根据 day 的值使用 switch...case 语句给出对应的输出，其与多重 if...else 语句的执行流程类似。



在使用 switch...case 语句时，读者一定要注意使用 break 语句，否则将因无法跳出分支而不能继续执行下去，从而导致输出错误。

在使用 switch 语句时应注意以下几点:

- 若 case 后面没有 break 语句, 则程序顺序执行后续的语句, 直到 switch 语句结束。这样就不能实现多分支选择。
- switch 语句只能对表达式的结果是否为固定的值进行判断, 而不能对其结果是否在某一区域进行判断, 这与多重 if...else 语句有所不同。
- 每个常量表达式的值不能相同, case 语句的排列顺序不影响执行结果。

通过上述几种分支语句实现选择结构的介绍, 读者可根据实际情况选择适当的语句。一般来说, 应尽量使用可读性强的语句。

4.2.5 应用示例

4.2.1 节~4.2.4 节详细讲解了 4 种选择结构的实现语句, 本节将通过一个较综合的示例回顾一下选择结构的实现。

【范例 4-12】选择结构应用示例。该范例求方程 $ax^2+bx+c=0$ 的根。该程序的实现需要接收用户输入 a、b、c, 并根据输入判断是否有实根求出, 实现代码如代码清单 4-12 所示。

代码清单 4-12

```

1  #include <iostream.h>                                //预处理文件
2  #include <math.h>
3  void main()
4  {
5      float a,b,c,d;                                    //定义浮点型变量
6      cout<<"a=";
7      cin>>a;                                           //接收用户的键盘输入
8      cout<<"b=";
9      cin>>b;                                           //接收键盘输入
10     cout<<"c=";
11     cin>>c;
12     d=b*b-4*a*c;                                       //计算求根公式内根号值
13     if (a==0)                                          //根据取值不同分别判断
14         cout<<"不是一元二次方程"<<endl;
15     else if (d>0)                                      //系数平方根大于 0
16     {
17         cout<<"方程有两个实根! "<<endl;
18         cout<<"x1="<<(-b+sqrt(d))/(2*a)<<endl;        //求根
19         cout<<"x2="<<(-b-sqrt(d))/(2*a)<<endl;
20     }
21     else if (d==0)
22     {
23         cout<<"方程有一个实根! "<<endl;              //求根
24         cout<<"x1=x2="<<-b/(2*a)<<endl;
25     }
26     else                                              //平方根小于 0
27         cout<<"方程无实根! "<<endl;
28 }
```

【执行结果】上述程序中使用了多重分支语句 if...else if...else 语句来实现一元二次方程的根在三种不同情况下的取值, 其执行结果如图 4-16 所示。

【范例解析】根据一元二次方程的求根公式 $(-b \pm \sqrt{b^2-4ac})/(2a)$, 需要先根据 (b^2-4ac) 中的值判断其有几个根。根据其取值的不同, 有三种不同情况: 当 (b^2-4ac) 大于 0 时方程有两个根, 等于 0 时有一个根, 小于 0 时则没有实根。范例 4-12 即采用了 if...else if...else 语句实现了这三种情况的判断, 使其根据取值不同, 执行不同的程序。



注意 在一个 if 或 else 语句内, 如果需要执行的代码多于 1 条语句, 就必须用 {} 将其所有语句包含起来, 构成复合语句。

为了便于读者理解上述范例, 此处给出该范例的执行流程图, 如图 4-17 所示。

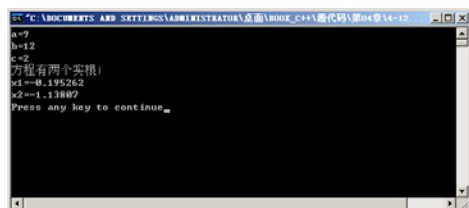


图 4-16 一元二次方程求解

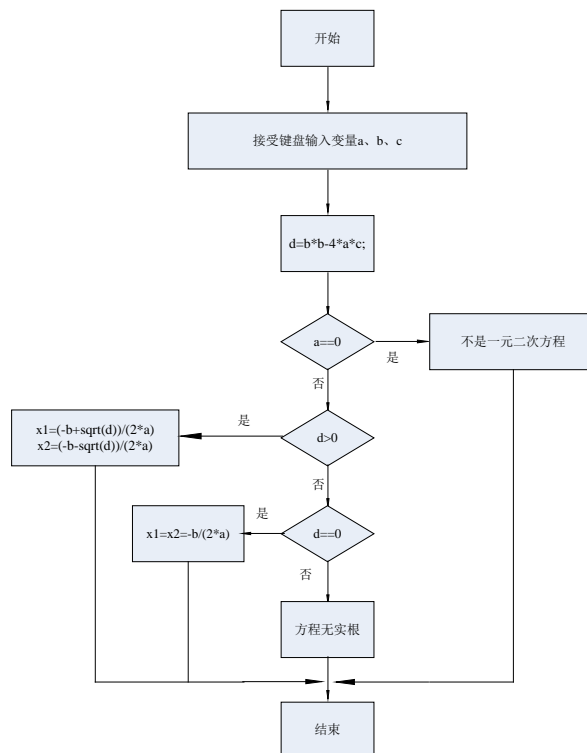


图 4-17 执行流程图

4.3 循环结构

循环结构是用来在指定的条件下多次重复执行同一组语句。在 C++ 中, 常用的循环语句形式主要有如下三种:

- for 语句
- while 语句
- do...while 语句

4.3.1 for 语句

for 语句是 C++ 中最常见的、功能最强的循环语句, 它既可用于循环次数确定的情况, 也可用于循环次数不确定而只给出循环结束条件的情况, 其说明语句的一般形式为:

```
for ( <表达式 1>; <表达式 2>; <表达式 3> )
    <语句>;
```

其中, 表达式 1 是对循环控制变量进行初始化, 表达式 2 是循环条件, 表达式 3 是对循环控制变量进行递增或递减。这 3

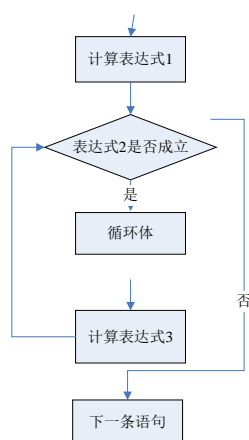


图 4-18 for 语句执行流程

个表达式都可以缺省,但分号不能省略。for 语句可以是单条语句,也可以是块语句,它是要被循环重复执行的程序段,故又称为循环体。其执行流程如图 4-18 所示。

【范例 4-13】for 语句的应用。该范例求自然数 100 内的所有偶数之和,即计算 $2+4+6+\cdots+100$ 的算术和。该范例与求 1~100 内所有自然数和相似,不同的是其只要求偶数,这就需要在赋初值时预先设置,并在控制循环变量时递增 2 即可,程序如代码清单 4-13 所示。

代码清单 4-13

```

1  #include <iostream.h>
2  void main()
3  {
4      int i;
5      int sum=0;                //初始化变量
6      for(i=2;i<=100;i=i+2)    //for 循环
7      {
8          sum+=i;              //递加
9      }
10     cout<<"100 内所有偶数和为:"<<sum<<endl;    //输出结果
11 }
```

【运行结果】在 Visual C++ 中编译上述程序代码,若编译无误使用快捷键 **【Ctrl+F5】** 执行该代码,其返回结果如图 4-19 所示。

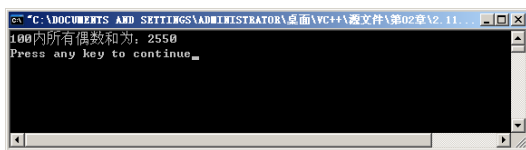


图 4-19 for 语句

【范例解析】范例 4-13 代码中,for 语句中的变量初始值为 2,这是第一个偶数,表达式 3 中的循环变量每次递增 2,保证 i 中的值都为偶数,循环条件为小于等于 100。

警告 for 语句中包含 3 个语句,这是必不可少的,即使其中某一个语句为空,“;”符号是不能省略的,否则编译系统将给出错误提示。

4.3.2 while 语句

while 语句是最简单的循环语句,它实际上是 for 语句的表达式 1 和表达式 3 为空的特殊情形,其说明语句的一般形式为:

```

while (<表达式> )
<语句>;
```

其中,表达式用来判定循环是否继续,当表达式条件成立时,执行循环体。while 语句要求能够在循环体内含改变物质循环条件表达式的值,以使表达式条件不成立时退出循环体。因此,其执行流程如图 4-20 所示。

使用 while 语句也同样可以实现类似 for 语句的循环。例如,范例 4-14 可以用 while 语句实现前面 for 语句实现的功能。

【范例 4-14】while 语句的应用。该范例将上述求 100

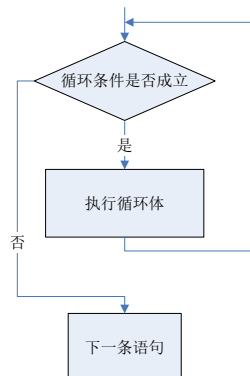


图 4-20 while 语句执行流程



以内所有偶数和的示例通过 while 语句来实现，代码如代码清单 4-14 所示。

代码清单 4-14

1	#include <iostream.h>	
2	void main()	
3	{	
4	int i;	
5	int sum=0;	//定义变量并初始化
6	i=2;	//初始化变量
7	while(i<=100)	//while 循环
8	{	
9	sum+=i;	//递增
10	i=i+2;	//变量 i 每次递增 2
11	}	
12	cout<<"100 内所有偶数和为: "<<sum<<endl;	//输出结果
13	}	

【运行结果】上述程序代码清单的执行结果与图 4-19 相同。

【范例解析】读者可以对比 for 语句和 while 语句，其不同就在于 for 语句将赋初值和循环变量的改变均集成在 for 语句中，并用“;”隔开，而 while 语句则写在语句块中作为更直观的语句，相对来说，while 语句的可读性更强，但 for 语句更为精练简洁。

警告 在 while 语句的循环体中，一定要包含如 i++之类的循环变量递增或递减的语句，使其经过若干次循环后不满足循环条件，从而跳出循环。否则，程序将陷入死循环中。

4.3.3 do...while 语句

do...while 语句是 while 语句的一种变化形式，其说明语句的一般形式为：

```
do
<语句>
while (<表达式> );
```

do...while 语句与 while 语句的主要区别是：do...while 语句的循环体至少被执行一次，而 while 语句先判断条件，有可能一次也不执行。其执行流程如图 4-21 所示。

【范例 4-15】do...while 语句的应用。该范例将上述求 100 以内偶数之和的示例采用 do...while 语句来实现，程序如代码清单 4-15 所示。

代码清单 4-15

1	#include <iostream.h>	
2	void main()	
3	{	
4	int i;	
5	int sum=0;	//定义变量并初始化
6	i=2;	//初始化变量
7	do	//do...while 循环
8	{	
9	sum+=i;	//递增
10	i=i+2;	//循环变量递增 2
11	}	

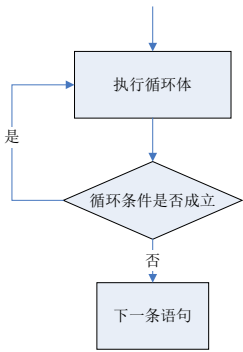


图 4-21 do...while 语句
执行流程

```

12     while(i<=100);           //循环条件
13     cout<<"100 内所有偶数和为: "<<sum<<endl;       //输出
14 }

```

【运行结果】执行上述程序，其执行结果与图 4-19 相同。

【范例解析】该代码与代码清单 4-14 的区别就在于 do...while 语句先执行循环体内的语句，后判断是否符合循环条件，而其他语句都类似。

总的来说，for 语句、while 语句和 do...while 语句都能够实现循环结构。但是在 C++ 中，最常用的还是 for 语句，这是因为它集成了三个语句在其中，写法精练。当然，读者可以根据不同的具体情况选择使用不同的语句。

4.3.4 多重循环

在实际的应用中，还有一种循环方式使用很广泛，这就是多重循环，也称为循环嵌套，其是指循环语句的循环体内又包含另一个循环语句。



注意 在多重循环中，循环嵌套的执行顺序是先执行最里层的循环语句，依次往外执行，最后执行最外层的循环。

例如，打印一个九九乘法表。读者可以理解，九九乘法表需要用两个数值进行运算，即乘数和被乘数，而这两个数在九九乘法表中是需要从 1~9 变化的，这就需要使用到多重循环了，其实现代码如代码清单 4-16 所示。

【范例 4-16】多重循环的实现，打印一个九九乘法表。

代码清单 4-16

```

1  #include<iostream.h>
2  void main( )
3  {
4      int bcs,cs;
5      for (bcs=1; bcs<=9; bcs++)           //bcs 表示行号，外循环
6      {
7          for (cs=1; cs<=bcs;cs++)         //cs 表示列号，内循环
8          {
9              cout<<bcs<<' '*<<cs<< '='<<bcs*cs<<' ' ; //输出格式
10             }
11             cout<<endl;                   //循环内换行
12         }
13     }

```

【运行结果】在 Visual C++ 中执行上述代码，其执行结果如图 4-22 所示。

【范例解析】范例 4-16 代码中，在 for 循环中又嵌套了一个 for 语句，作为其子循环。执行该程序时，先执行里层的子循环，再执行外层循环。

多重循环在具体的应用中使用非常广泛，为了让读者更好地理解多重循环，这里给出范例 4-16 的程序流程图，如图 4-23 所示。



提示 在循环嵌套中，并非一定要同种结构的循环才能进行嵌套。for 循环里面可以嵌套 while/do...while 循环，while 循环里也可嵌套 for/do...while 循环。循环体内可以嵌套多个内循环，内循环里还可以嵌套内循环。

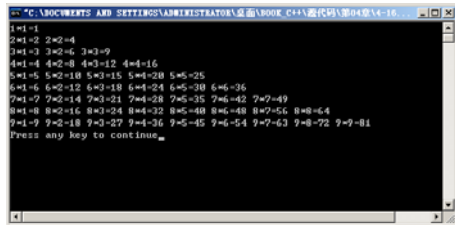


图 4-22 多重循环

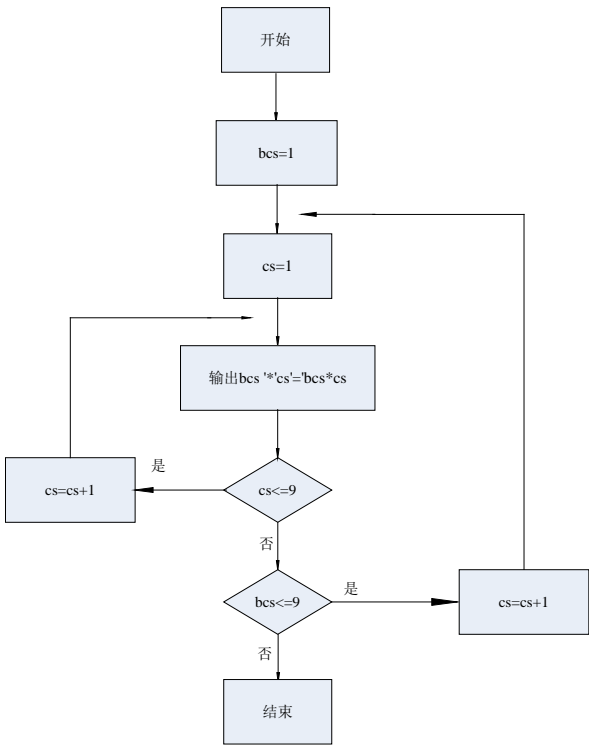


图 4-23 执行流程

4.3.5 应用示例

4.3.1~4.3.4 节的内容详细介绍了三个实现循环结构的语句，以及多重循环的实现。下面通过一个具体示例回顾循环结构的实现。下列程序求 1000 内的所有水仙花数。所谓水仙花数，是指一个三位数，其各位数字立方和等于该数字本身。

【范例 4-17】循环结构应用示例。在该范例中，需要分开一个数的百位、十位和个位，然后求出其立方和是否与该数字本身相等，如相等则输出，否则继续寻找下一个数，其实现程序如代码清单 4-17 所示。

代码清单 4-17

```
1  #include <iostream.h>
2  void main()
3  {
4      int i,j,k,n,m;
5      for (i=1;i<=9;i++)//百位数从 1 到 9 变化，不能取 0，如果百位为 0 就不是三位数了
6      {
7          for (j=0;j<=9;j++)
8              //十位从 0 到 9 变化，允许取 0
9              {
10                 for(k=0;k<=9;k++)
11                     //个位从 0 到 9 变化，允许取 0
12                     {
13                         n=i*100+j*10+k;
14                         //这个数等于百位数乘以 100 加上十位数乘 10 加上个位数(乘以 1 省略)
15                         m=i*i*i+j*j*j+k*k*k;
16                         //百位上数的立方加上十位上数的立方加上个位上数的立方
```

```

13             if(n==m) cout<<n<<" ";
                //这是一个三位数是否为水仙花数的条件，即水仙花数是各位数字立方和等
                //于该数字本身
14         }
15     }
16 }
17     cout<<endl;
18 }

```

【运行结果】范例 4-17 的代码在 Visual C++ 中的执行，结果如图 4-24 所示。

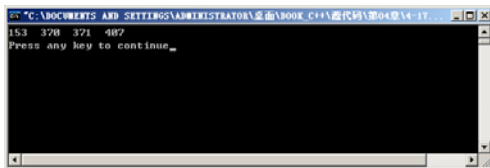


图 4-24 求解水仙花数

【范例解析】范例 4-17 的代码中，使用了一个三重循环，分别取一个数的百位、十位和个位， n 为这个数本身， m 为该数各位的立方和，如 $n==m$ 成立，则 n 位水仙花数，将其输出，并找下一个数，直至百位数大于 9 即结束整个循环。

4.4 转向语句

转向语句是 C++ 中用来实现无条件转移的语句。常用的转向语句有如下 4 种。

- **break 语句：**break 语句又称跳出语句，用来结束循环结构，然后执行循环体后面的语句，其说明语句的一般形式为：

break;

break 语句也可以作为 switch 语句的出口，用于退出 case 语句。

- **continue 语句：**continue 语句又称继续语句，可用来跳出本次循环而进入下一次循环，其说明语句的一般形式如下：

continue;

continue 语句与 break 语句的主要区别是，continue 语句是根据条件判断只结束本次循环，不结束整个循环结构；而 break 语句不进行判断，结束整个循环结构，然后执行循环体后面的语句。

- **goto 语句：**goto 语句又称转向语句，用来将程序无条件跳转到指定的标号语句处，其说明语句的一般形式如下：

goto<标号>;

其中标号是一个标识符，放在语句的最前面，其说明语句的一般形式为：

<标号>: <语句>



注意 使用 goto 语句将使程序结构不清晰，可读性低。一般来说，在结构化程序设计中应尽量少用或不用 goto 语句。

- **return 语句：**return 语句又称返回语句，可用来停止执行当前函数，转而执行调用该函数后面的语句，其说明语句的一般形式如下：

return<表达式>;



表达式可以是任何类型的变量，也可以是 void 型。需要注意的是，所返回表达式的类型必须与函数的类型一致。

【范例 4-18】转向语句的应用。该范例求出 100 以内的所有素数。所谓素数是指大于 2 且只能被 1 或本身整除的整数。判断素数的算法为：对于 i，只要其能够被 2~i-1 中任一个数整除，则 i 不是素数。其实现代码如代码清单 4-18 所示。

代码清单 4-18

```
1  #include <iostream.h>           //预处理文件
2  int main(int argc,char * argv[]) //主函数
3  {
4      const n=100;
5      int i,j;                     //定义常量、变量
6      for(i=2;i<=n;i++)           //外循环
7      {
8          int flag=1;              //定义标志, flag 为 1 时为素数
9          for(j=2;j<i;j++)         //内循环
10             if(i%j==0)           //判断 i 是否素数
11             {
12                 flag=0;
13                 break;           //如果 i 能够被 j 整除, 则 i 不是素数, 退出本次循环
14             }
15             if(flag==1)
16                 cout<<i<<" ";   //依次显示素数
17         }
18         cout<<endl;              //输出换行
19         return 0;
20     }
```

【运行结果】在 Visual C++ 6.0 中执行上述代码，执行结果如图 4-25 所示。

【范例解析】在范例 4-18 的程序代码中，使用了两重循环，前一个循环用于取 2~100 内的所有数字，后一个循环用于判断取到的数值是否是素数。

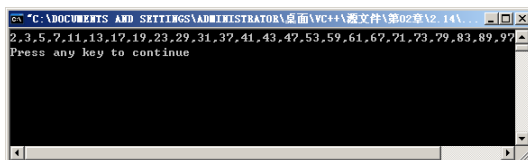


图 4-25 求素数

4.5 小结

本章主要介绍了 C++ 中实现程序控制结构的各种语句。C++ 的程序控制结构与大多数语言相同，分别为顺序结构、选择结构和循环结构。针对其中每一种结构，本章都给出了各自的实现语句，并给出了一个较为综合的应用示例供读者理解。在实际的应用中，这三种基本结构的使用非常频繁，读者应熟练掌握其实现语句和基本思想。本章最后简要介绍了 C++ 中的 4 个转向语句，这些语句也可以控制程序的流程，但都有些使用限制，读者要仔细理解。

4.6 习题

1. C++ 程序中有如下语句：

```
n=(i=2,++i)
```

该语句是否是合法的? 如果合法, 运行结束后 n 和 i 的值分别为多少?

【解答】该习题考查逗号运算符和赋值语句。上述表达式语句是合法的, 其总体上是一个赋值语句, 赋值号右边是一个逗号表达式, 逗号表达式先计算第一个表达式, 最终结果为括号中最后一个表达式的值。因此运行结束后 i 的值为 3, 而运行 $n=3$ 语句后, n 的值也为 3。

2. 编写一个 C++ 程序, 要求接收用户输入的一个包含 5 位小数的浮点数, 通过精度控制输出该浮点数有效数字为 3 位、4 位和 5 位时的数值。例如, 输入一个浮点数 15.41532, 运行程序后将分别显示如图 4-26 所示结果。

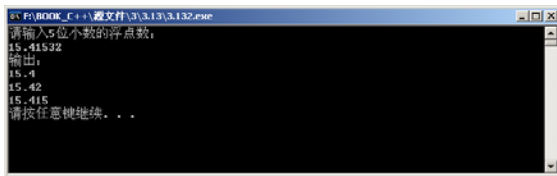


图 4-26 控制输出精度

【解答】该习题主要考查使用 `cout` 输出流控制精度。程序要求接收用户输入, 因此需声明一个浮点型变量, 输出指定的有效位数, 只需使用 `cout` 输出流的精度控制成员函数 `setprecision` 即可。需要注意的是, 输出有效位数时, 对最后一位有效位数要进行四舍五入操作。其简要代码如下所示。

```
double a;
cout<<"请输入 5 位小数的浮点数: "<<endl;
cin>>a;
cout<<"输出: "<<endl;
cout<<setprecision(3)<<a<<endl;           //输出精度为 3
cout<<setprecision(4)<<a<<endl;           //输出精度为 4
cout<<setprecision(5)<<a<<endl;           //输出精度为 5
```

3. 已知 $\text{int } x=10, y=20, z=30$, 以下语句执行后 x, y, z 的值将分别是多少?

```
if(x>y)
z=x;x=y;y=z;
```

【解答】该习题主要考查 `if` 语句。在上述语句中, 首先判断关系条件 $x>y$ 的值是否为 `true`, 此处将 x 和 y 的值分别代入: $x=10, y=20$, 因此 $x>y$ 是不成立的, 因此 `if` 语句下面的第一条语句不会执行, 即语句 `z=x` 不会执行, 而其后的两条语句 `x=y` 和 `y=z` 语句将会被执行。因此, 执行该语句段后, x, y 和 z 的值分别为 20, 30 和 30。

4. 编写一个 C++ 程序, 从键盘输入三角形的三边长, 判断出这三边能否构成三角形。例如, 输入三边分别为 2, 3, 4, 其返回结果如图 4-27 所示。

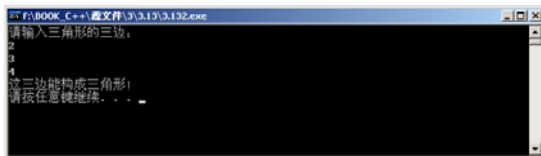


图 4-27 判断三角形构成

【解答】该习题主要考查 `if...else` 语句。判断三边是否能够构成三角形, 需要满足如下条件: 即任意两边的和要大于另一边, 即 $a+b>c$, 同时 $a+c>b$, 同时 $b+c>a$ 。根据该定理, 这三个条件要同时成立, 因此其关系为与关系。在 `if` 语句后的条件表达式要表示为: $(a+b>c) \&\& (a+c>b) \&\& (b+c>a)$, 满足该条件即可构成三角形, 否则不能构成。其简要代码如下所示。



```
int a,b,c;
cout<<"请输入三角形的三边: "<<endl;
cin>>a>>b>>c;
if ((a+b>c) && (a+c)>b && (b+c)>a )
    cout<<"这三边能构成三角形! "<<endl;
else
    cout<<"这三边不能构成三角形! "<<endl;
```

5. 符号函数的实现在许多程序中是常见的, 用多种选择语句编程实现符号函数。当 $x < 0$ 时 $\text{sgn}(x) = -1$, 当 $x > 0$ 时 $\text{sgn}(x) = +1$, 当 $x = 0$ 时 $\text{sgn}(x) = 0$ 。例如, 当用户输入 x 的值为 10 时, 其输出结果为 1, 输入 x 的值为 -10 时输出结果为 -1, 输入 x 的值为 0 时输出结果为 0, 如图 4-28 所示。

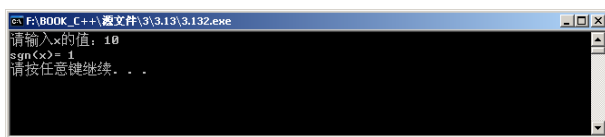


图 4-28 符号函数的实现

【解答】该习题主要考查 `if...else if...else` 语句的应用。读者可以看到, 上述程序有 3 种可能情况, 即输入的 x 有大于 0、小于 0 和等于 0 这 3 种情况。此处需要使用 `if...else if...else` 语句判断这三种情况, 并输出对应的值。其简要代码如下所示。

```
float x;
int y;
cout<<"请输入x的值: ";
cin>>x;
if (x>0)
    y=1;
else if (x==0)
    y=0;
else
    y=-1;
cout<<"sgn(x)= "<<y<<endl;
```

6. 当执行以下程序时, 循环体将被执行多少次?

```
k=1;
do{
    k=k*k;
}while(!k);
```

【解答】该习题主要考查 `do...while` 语句的执行情况。`do...while` 语句不管循环条件是否满足, 都至少会执行一次循环体, 执行一次后判断循环条件是否成立。此处 k 的初值为 1, 执行完语句 `k=k*k;` 后, k 的值仍然为 1。因此, 循环条件 `!k` 的值为 `false`, 该循环不会再继续下去, 从而退出循环。因此, 该循环体被执行了一次。

7. 有 1、2、3、4 个数字, 能组成多少个互不相同且无重复数字的三位数? 编写一个程序输出所有无重复数字的三位数。

【解答】该题目主要考查嵌套循环和多个表达式的逻辑判断。为了方便理解, 分析题目要求:

- (1) 由于是三位数, 所以假设百位数、十位数、个位数依次为 x 、 y 和 z 。
- (2) 由于从 1、2、3 和 4 选取, 所以 $1 \leq x \leq 4$ 、 $1 \leq y \leq 4$ 和 $1 \leq z \leq 4$ 。
- (3) 由于互不重复, 所以 $x \neq y$, 且 $y \neq z$, 且 $z \neq x$ 。

所以, 只需要让 x 、 y 和 z 依次取值, 然后判断条件, 只要满足条件就可以。其简要代码如下所示。



```
for(x=1;x<=4;x++)
{
    for(y=1;y<=4;y++)
    {
        for(z=1;z<=4;z++)
        {
            if(x!=y&&y!=z&&z!=x)    cout<<(x*100+y*10+z);
        }
    }
}
```

8. 编写一个程序, 求 $12+22+32+42+\cdots+202$ 的值。

【解答】该题目考查的是读者的归纳能力和循环语句使用能力。首先分析该表达式的特点。该表达式是计算数十个数字的和。为了方便读者找出规律, 我们将所有的数字都列出, 如表 4-2 所示。

表 4-2 所有累加的值

12	22	32	42	52	62	72	82	92	102
112	122	132	142	152	162	172	182	192	202

从这些数字中可以总结出以下规范: 每个数字都比前一个数字大 10; 这些数字个数为 20。所以, 第一个数字可以表示为 $a_1=2+10=12$; 第二个数字可以表示为 $a_2=a_1+10=22$; 第三个数字可以表示为 $a_3=a_2+10=32\cdots a_i=a_{i-1}+10\cdots a_{20}=a_{19}+10=202$ 。其简要代码如下所示。

```
int a,sum;
a=2;
for(i=1;i<21;i++)
{
    a=a+10;
    sum=sum+a;
}
```

9. 编写一个 C++ 程序, 找出 200 内能被 7 整除的所有自然数, 并将其输出到用户屏幕。

【解答】该习题主要考查 continue 语句的应用。continue 语句用于结束本次循环, 跳到下一次循环。在该习题中, 通过一个循环在 1~200 之间依次进行查找, 能够被 7 整除则输出, 否则使用 continue 语句结束本次循环, 继续查找。其简要代码如下所示。

```
for (int a=17;a<=100;a++)
{
    if (a%17!=0)
        continue;
    cout<<a<<" ";
}
cout<<endl;
```

第二篇 C++面向过程设计篇

第5章 函数

不论是在哪一种程序设计语言中，函数都是一个重要的组成部分。在 C++ 中，函数是一个能完成某一独立功能的子程序，或者说是程序模块。函数就是对复杂问题的一种“自顶向下，逐步求精”思想的体现。用户可以将一个大而复杂的程序分解为若干个相对独立而且功能单一的小块程序（函数）进行编写，并通过在各个函数之间进行调用来实现总体的功能。本章将主要介绍 C++ 中函数的声明、参数传递，以及调用等基本操作。

以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为一个学习的参照。

- 掌握 C++ 中函数的声明与定义。
- 熟练掌握函数的参数、原型和返回值，以及如何在程序中调用函数。
- 了解 C++ 中函数的重载。

5.1 定义函数

在结构化程序设计中，通常把一个大的程序分成若干个模块，每一个模块完成一个或多个特定功能。每一个模块是相对独立的，却又具有通用性，可供本程序或其他程序调用。这种模块化的设计思想有利于多人协作共同开发程序，函数就是这种设计思想的产物。在结构化程序设计中，函数及其相互之间的调用构成了整个应用程序，如图 5-1 所示。

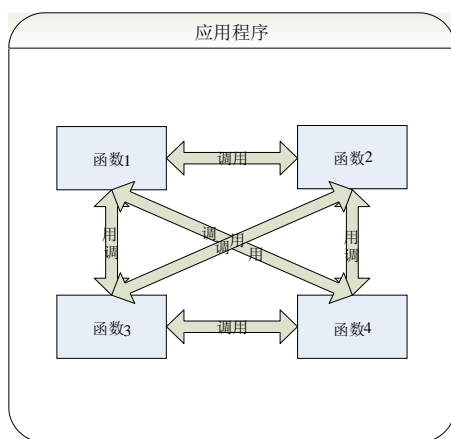


图 5-1 应用程序与函数

5.1.1 函数概述

在具体讲解如何定义函数和调用函数前，先来了解一下在程序设计语言中为什么要引入函数这个概念。前面说到了，函数有利于模块化，这对于结构化程序是非常方便的。此外，在程序中使用函数还有如下的优势：

- 程序可读性好。
- 程序易于查错和修改。
- 便于分工编写，分阶段调试。

- 各个函数之间接口清晰，便于相互间交换信息和使用。
- 节省程序代码和存储空间。
- 减少用户总的工作量。
- 成为实现结构程序设计思想的重要工具。
- 能够扩充语言和计算机的原设计能力。
- 便于验证程序正确性。

简单地讲，设计一个 C++ 程序的过程，实际上就是编写函数的过程，至少也要编写一个

main()函数。执行 C++ 程序，也就是执行相应的 main() 函数，即从 main() 函数的第一个 “{” 开始，依次执行后面的语句，直到最后一个 “}” 为止。

如果在执行过程中遇到其他的函数，则调用其他函数。调用完后，返回到刚才调用函数语句的下一条语句继续执行。而其他函数也只有在执行 main() 函数的过程中被调用时才会执行。函数可以被一个函数调用，也可以调用另一个函数，它们之间可以存在着调用上的嵌套关系。C++ 函数是一个独立完成某个功能的语句块，函数与函数之间通过输入和输出来联系。



警告 C++ 不允许函数的定义嵌套，即在函数定义中再定义一个函数是非法的。

5.1.2 定义函数

在 C++ 程序中调用函数之前，首先要对函数进行定义。如果调用此函数在前，函数定义在后，就会产生编译错误。为了使函数的调用不受函数定义位置的影响，可以在调用函数前进行函数的定义。这样，不管函数是在哪里定义的，只要在调用前进行了函数的定义，就可以保证函数调用的合法性。

函数定义的一般形式如下：

```
返回类型 函数名(参数列表)
{
    ...
    函数体
}
```

一般来说，函数的定义需要包括以下几个部分。

- **函数名：**即一个符合 C++ 语法要求的标识符。定义函数名与定义变量名的规则是一样的，但应尽量避免用下画线开头，因为编译器常常定义一些下画线开头的变量或函数。并且函数名应尽可能反映函数的功能，其常常由几个单词组成。如 Visual C++ 中的按下鼠标左键的响应函数为 OnLButtonDown，这样就较好地反映了函数的功能。此外，在函数名后面必须跟一对圆括号 “()”，用来将函数名与变量名或其他用户自定义的标识符区分开来。在括号中可以没有任何信息，也可以包含形式参数表。C++ 程序通过使用这个函数名和实参表可以调用该函数。
- **参数列表：**参数也即 0 个或多个变量，写在函数名后面的一对圆括号内，用于向函数传送数值或从函数带回数值，其不同于变量定义，每一个参数都有自己的类型。当参数列表中的参数多于一个时，其前后两个参数说明项之间必须用逗号分开。如果参数列表中参数个数为 0，称为无参函数。
- **返回类型：**即指定函数用 return 返回的函数值的类型，如果函数没有返回值，返回类型应为 void。每个函数都有类型，如果在函数定义时没有明确指定类型，则默认类型为 int。
- **函数体：**花括号中的语句称为函数体，一个函数的功能，通过函数体中的语句来完成。函数体是函数的主体部分，其一般是一条复合语句，其以 “{” 开始，到 “}” 结束，中间为一条或若干条 C++ 语句，用于实现函数执行的功能。

例如，下面定义一个函数 dec，其功能是返回两个数的差值，其函数的定义语句如下：

```
int dec(int x,int y)
{
    return(x-y);
}
```

在上述函数的定义中，函数名为 dec，参数列表为 int x、int y，返回类型为 int，函数体为 return(x-y); 这条语句。此外，如果需要定义一个无返回值的函数，只需将函数名前的返回类型设置为 void 即可。例如，下列函数 out 用于输出两个数的差值及和值。



```
void out(int x,int y)                //用 void 表示无返回值
{
    cout<<x-y;
    cout<<x+y;
}
```



注意 在函数定义的函数体中可以定义其他变量，或使用前面章节讲解到的流程控制语句，用以完成函数的功能。

例如，下面定义的函数 `max` 返回两个数之间的较大数。

```
int max(int a,int b)
{
    int t;
    if(a>b) t=a;
    else t=b;
    return t;
}
```

除此之外，C++中不允许函数定义嵌套，即在函数定义中再定义一个函数是非法的。一个函数只能定义在别的函数的外部，函数定义之间都是平行的，互相独立的。例如：下面的代码在主函数中嵌套了一个名为 `f()` 函数定义，这是非法的。

```
void main()
{
    void f()
    {
        //...
    }
}
```

5.1.3 应用示例

鉴于函数定义的重要性，为了使读者能更好地理解函数的定义，本节给出一个函数定义的应用示例并在主函数中调用，读者就可以理解函数的功能了。

【范例 5-1】函数的定义。该范例定义一个函数 `func`，该函数判断指定的参数与 0 的比较结果，大于 0 则返回值 1，等于 0 则返回 0，小于 0 则返回 -1。并在主函数 `main()` 中调用该函数，使读者可以看到该函数的执行结果。程序段如代码清单 5-1 所示。

代码清单 5-1

```
1  #include <iostream.h>
2  int func(int n)                //声明函数 func，返回类型为 int，参数为 int n
3  {                              //函数体
4      if(n>0)                    //n>0 成立
5          return 1;
6      else if(n==0)              //n=0 成立
7          return 0;
8      else                       //n<0 成立
9          return -1;
10 }
11 void main()
12 {
13     int n;                      //定义变量
14     cout<<"Please input n:"<<endl;
15     cin>>n;                    //接收用户输入
16     cout<<"\nthe result:"<<func(n)<<endl; //调用函数
17 }
```

【运行结果】在 Visual C++ 中运行上述程序，其执行结果如图 5-2 所示。

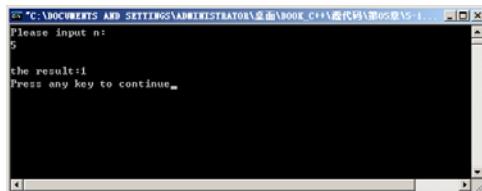


图 5-2 函数定义

【范例解析】上述代码中，首先使用函数的定义格式定义了一个函数 `func`，该函数带有一个整型参数，该参数称为形式参数，在调用该函数的时候该参数将会被实际参数的参数所取代，关于形参与实参在后续章节中还将详细介绍。

5.2 函数参数及原型

在 5.1 节中介绍到函数一般都带有参数列表，这个参数列表称为形式参数，而在调用函数时，实际的参数和形式参数将会有数据传输。

函数原型也称函数声明或函数模型。在主调函数中，如果要调用另一个函数，则须在本函数或本文件中的开头将要被调用的函数事先作一声明。声明函数就是告诉编译器函数的返回类型、名称和形参表构成，以便编译系统对函数的调用进行检查。

5.2.1 函数的参数及返回值

在介绍函数定义的一般格式中，提到了函数的参数列表，即在函数名后的一对圆括号()内的参数，称之为形式参数，简称形参；而在主函数中调用该函数时指定的参数称为实际参数，也称实参。简单地说，被调用函数与主调用函数之间的通信可以通过参数的传递来实现。

如代码清单 5-1 中所示，函数定义语句 `int func(int n)` 中 `int n` 即定义了一个形式参数，而在主函数的调用语句 `cout<<"\nthe result:"<<func(n)<<endl;` 中，`func(n)` 中的 `n` 即为实际参数。



注意 实际参数必须与形式参数的个数相同、数据类型相同，而且其对应顺序必须为一一对应，这在后续的参数传递中还将重点讲解。

获得函数的返回值是调用函数的目的，也就是说，在主函数中调用某一函数的目的就是要获得被调用函数的返回值，以便在主函数中使用。对于函数与其返回值的关系，就如同加工厂中机器与产品之间的关系，工人将原材料放入到加工机器中，加工后出来的是产品，如图 5-3 所示。

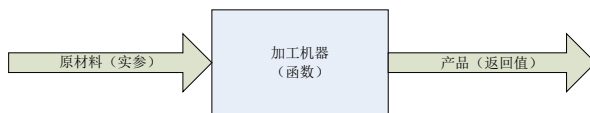


图 5-3 函数返回值

对于函数也可以这样理解，加工机器就是函数，原材料就是实参，产品就是函数的返回值了，调用函数最终就是为了获得产品。例如，下列定义的函数 `swap()`：

```
int swap(int a,int b)
{
    int temp;
    temp = a;a=b;b=temp;
    return 0;
}
```



上述函数中，swap()的括号中定义的整型变量 a、b 就是函数的参数，该函数返回一个整型值，在函数中返回为 0。

5.2.2 函数原型

C++中，如果在使用函数前没有对函数进行定义，则必须对函数进行声明。函数原型声明用来指出函数的名称、类型和参数，其说明语句的一般形式为：

```
[<属性说明>]<函数类型><函数名>(<参数>);
```

其中各部分的含义如下：

- 属性说明可以默认，一般为 inline（内联函数）、static（静态函数）、virtual（虚函数）、friend（友元函数）等这几个关键词之一。
- 函数类型是指函数返回值的类型。
- 函数名为一个 C++ 标识符。
- 参数，也称形式参数（形参），要求形参在函数原型声明、定义和调用时数据类型、个数、顺序一致，各形参名可以不同。

函数原型声明一般出现在程序中函数的调用之前，其目的是使编译器知道该函数的各种属性，包括返回值类型、形参的个数和类型等，以便于检查函数调用的合法性。事实上，前面 5.1.2 节中讲解的函数定义包括函数原型声明及函数体。例如，如果有一个函数的定义如下：

```
double func1(double a, int b, float c)
{
    函数体
}
```

那么正确完整的函数声明应为：

```
double func1(double x, int y, float z);    //末尾要加上分号
```

也可以写为如下形式：

```
double    func1(double,int,float);        //函数声明中省略了形参名
```

或写为如下形式：

```
double func1(double a, int b, float c);    //函数声明中的形参名与函数定义中的形参名不同
```



提示 读者可以看出，除了需在函数声明的末尾加上一个分号“;”之外，其他的内容与函数定义中的第一行（称函数头）的内容一样。

但是，读者需要注意，函数声明是不能省略参数的类型的，也不能省略函数的返回值类型，或者改变参数顺序，因此，下列函数的原型声明都是错误的：

```
double func1(x,y,z);                //函数声明中省略了形参类型
func1(double x, int y, float z);    //函数声明中省略了函数类型
double func1(int y, float z, double x); //函数声明中形参顺序调换了
```

5.2.3 main()函数

每个 C++ 程序都必须要有个 main() 函数，main() 函数也称为主函数，是 C++ 程序中最重要的函数，所有完整可运行的 C++ 程序都必须有一个唯一的 main() 函数。读者在使用 main() 函数之前，需要对以下两点注意事项有一个简单的了解，才能更好地使用 main() 函数。

- 对于一些简单的问题，只用一个 main() 函数即可，程序的全部处理语句都放在其中。
- 对于一些复杂的问题，需要进行模块化设计，即把一个复杂的问题分解成若干个相对

简单的一些子问题。每个子问题由一个和多个函数来处理，而 `main()` 函数负责总控，调用相应的函数。

`main()` 函数是一个特殊的函数。其中“`main`”是函数名，与其他函数一样，该函数也可以有返回值和参数表。在后续章节中将讨论 `main()` 函数的参数和返回值的使用问题，以下给出 `main()` 函数的最简单形式：

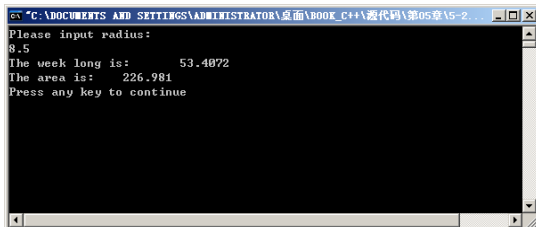
```
main()                                //主函数名
{                                     //函数体起始符
    变量声明语句
    执行语句
}                                     //函数体终止符
```

【范例 5-2】`main()` 函数的应用。该范例完整地体现了 C++ 程序的结构，读者可根据注释仔细理解，其代码如代码清单 5-2 所示。

代码清单 5-2

```
1  #include <iostream.h>
2  const double PI=3.1416;           //声明常量(只读变量)PI 为 3.1416
3  double fcir_l(double);           //声明圆周长计算函数
4  double fcir_s(double);           //声明圆面积计算函数
5  void main()                       //main()函数
6  {
7      double radius,cir_l,cir_s;
8      cout<<"Please input radius:"<<endl;
9      cin>>radius;                  //接收键盘输入
10     cir_l=fcir_l(radius);          //调用计算周长函数
11     cout<<"The week long is:"<<"\t"<<cir_l<<endl; //输出周长
12     cir_s=fcir_s(radius);          //调用计算面积函数
13     cout<<"The area is:"<<"\t"<<cir_s<<endl; //输出面积
14 }
15 double fcir_l(double r)           //定义圆周长计算函数
16 {
17     double cir_l;
18     cir_l=2*PI*r;                  //计算圆周长
19     return cir_l;                  //返回计算结果
20 }
21 double fcir_s(double r)           //定义圆面积计算函数
22 {
23     double cir_s;
24     cir_s=PI*r*r;                  //计算圆面积
25     return cir_s;                  //返回结果
26 }
```

【运行结果】上述代码在 Visual C++ 中执行，其运行结果如图 5-4 所示。

图 5-4 `main()` 函数

【范例解析】上述程序段首先声明了两个函数原型，再在 `main()` 函数中调用，而将两个函



数的定义写在 `main()` 函数之后，但由于在 `main()` 函数中调用其之前已经声明过，因此这是允许的。事实上，这也是 C++ 程序较为普遍的写法。



注意 `main()` 函数是 C++ 程序中唯一可以直接运行的函数，其他函数都直接或间接由其调用执行。C++ 程序的执行开始于 `main()` 函数，一个结构良好的 C++ 程序也应该结束于该函数。

范例 5-2 的代码清单 5-2 是 C++ 中一个较为普遍的示例，读者应仔细理解其写法。此外，在使用 C++ 编写程序时，应注意如下的事项：

- C++ 程序中每个语句都以分号 “;” 结束，此处的分号为英文半角字符，在输入 C++ 源程序时一定要注意不能写成全角，否则编译将不能通过。
- 函数的定义中使用了花括号 “{” 和 “}”，花括号作为函数体的开始和结束标记。使用花括号也是 C++ 的特点之一，但一定要配对使用。
- `main()` 函数是主函数，其作用是声明变量，接收用户从键盘输入的字符，如上面示例在主函数中调用函数计算圆的周长和面积，最后输出结果。
- `main()` 函数中的 `cout` 是 C++ 语言系统提供的用于控制显示输出的对象，通过它可以输出各种类型的数据，这在前面章节中已经介绍过了。
- `main()` 函数中的 `cin` 是 C++ 语言系统提供的用于控制数据输入的对象，通过它可以输入各种类型的数据，这在前面章节中已经介绍过了。
- 程序中以 “//” 开始的部分为注解。
- 程序最开始的一行，即 `#include <iostream.h>` 为 C++ 的预处理命令，其作用是将头文件 `iostream.h` 中的内容插入进来，该文件包含了键盘输入和显示输出的有关信息。



提示 C++ 中，`#include` 命令使用较多，其称为预处理指令，是 C++ 系统提供的一组指导编译的指令之一，在后面的章节中还将具体讲解。

5.2.4 带参数的 `main()` 函数

在前面的程序中 `main()` 函数都没有带参数，因此其圆括号中都是空的。实际上，`main()` 函数是可以带参数的。在 `main()` 函数中允许带两个参数，一个为 `argc`，整型数据类型，另一个是指向字符型的指针数组 `argv[]`。这两个参数在 `main()` 函数头部声明的格式为：

```
int main(int argc, char *argv[ ])
```

其中，整型参数 `argc` 表示命令行中字符串的个数，指针数组 `argv[]` 指向命令行中的各个字符串。这两个参数可以用任何合法的标识符命名，但习惯上采用 `argc` 和 `argv` 表示。带参数的 `main()` 函数一般能在调用其时追加参数，如 DOS 命令中 `dir /s` 一样，其中的 “/s” 就是参数。

【范例 5-3】带参数的 `main()` 函数的应用。该范例通过一个带命令行参数的示例代码来查看其具体的使用方法和功能，该范例实现输出当前文件的路径，其代码如代码清单 5-3 所示。

代码清单 5-3

```
1  #include <iostream.h>                                //包含输入/输出头文件
2  int main(int argc, char *argv[])                      //带参数的主函数
3  {
4      cout<<"The program file is : "<<argv[0]<<endl;    //输出信息
5      for (int k=1; k<argc; k++)
6          cout<<"argv[ "<<k<<" ]="<<argv[k]<<" ";    //循环输出第二个参数中的信息
7      cout<<endl;                                        //输出换行
```

```

8     return 0;
9 }

```

【运行结果】上述程序详细地说明了 argc 参数和 argv[] 数组中存储的内容,如果直接在 Visual C++ 的环境中运行该程序,其结果如图 5-5 所示。

但是,这并不符合带参数的 main() 函数要实现的功能,带参数的 main() 函数是要像 DOS 命令一样能够根据参数执行的。因此,可以在操作系统的 DOS 环境下执行该程序。在 DOS 下进入到目标可执行文件的路径后,输入文件名和参数。在该示例中,文件名为 5-3,输入参数为 abc 和 xyz,其运行结果如图 5-6 所示。

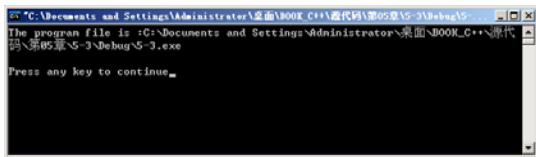


图 5-5 带参数的 main() 函数执行

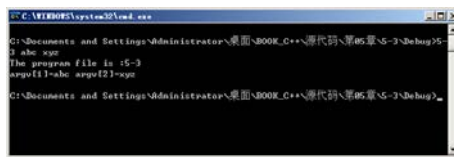


图 5-6 带参数的 main() 函数执行

【范例解析】根据图 5-6 的执行结果,读者可以看出,用户输入了两个字符串,因此 argc 的值为 2,在字符串数组 argv[] 中将这两个字符串分别放入 argv[1] 和 argv[2] 中,而 argv[0] 中存储的是该程序的当前路径。至此,读者应该明白这两个参数的作用了。

5.3 调用函数

函数声明后,在其他程序中即可对其进行调用了。一般来说,C++ 程序都是从主函数 main() 开始执行,当执行到函数调用语句时,就会转去执行调用函数,执行后仍然返回到主函数,直至程序结束。当调用一个函数时,整个调用过程分为三步进行:第一步是参数传递,第二步是函数体执行,第三步是返回,即返回到函数调用表达式的位置。

5.3.1 函数调用格式

在具体讲解调用函数的其他内容前,先简要了解一下函数调用的格式。一般来说,函数调用的形式为:

<函数名>(<实参表>)



注意 实参应该与函数定义中的形参表中的形参一一对应,即个数相等、次序一致且对应参数的数据类型相同或相容。每个实参是一个表达式,并且必须有确定的值。

例如,下面的函数调用,其实参都不同:

```

func(25)           //实参是一个整数
func(x)           //实参是一个变量
func(a,2*b+3)     //第一个为变量,第二个为运算表达式
func(sin(x),'@')  //第一个为函数调用表达式,第二个为字符常量
func(&d,*p,x/y-1) //分别为取地址运算、间接访问和一般运算表达式

```

一般来说,常见的函数调用方式有下列两种:

- 将函数调用作为一条表达式语句使用,只要求函数完成一定的操作,而不使用其返回值。若函数调用带有返回值,则这个值将会自动丢失。例如:

```
max(3,5);
```
- 对于具有返回值的函数来说,把函数调用语句看作语句的一部分,使用函数的返回值参与相应的运算或执行相应的操作。例如:



```
int a=max(3,5);
int a=max(3,5)+1;
cout<<max(3,5)<<endl;
if(f1(a,b)) cout<<"true"<<endl;
int a=2; a=max(max(a,3),5);
```

需要注意的是，在函数原型声明中的参数称为形式参数（形参），而在函数调用中的参数称为实际参数（实参），实参是实际调用函数时所给定的常量、变量或表达式，它必须有具体的值。在主调程序和被调用的函数之间数据的传递是通过参数表来实现的。

此外，C++有两种不同的函数：库函数和自定义函数。库函数是C++系统提供的标准函数，用户一般不必自己定义，需要时直接调用即可。在调用库函数时，一般在文件的开头通过#include宏命令引用库函数对应的原型声明头文件，自定义函数则是根据程序的需要由用户自行定义。



提示 在实际程序中，#include 命令后一般使用<>符号来调用库函数，使用“”符号来调用自定义函数。

5.3.2 传值调用

函数参数传递机制问题，本质上是调用函数（过程）和被调用函数（过程）在调用发生时进行通信的方法问题。前面内容提到了，函数调用的第一步就是传递参数。参数传递称为“虚实结合”，即实参向形参传递信息，使形参具有确切的含义（即具有对应的存储空间和初值）。根据参数传递的方式，函数调用可分为两种不同的方式，一种是按值传递，另一种是地址传递或引用传递。本节将介绍传值调用的实现，下一节将对传地址调用做详细讲解。

传值调用即其参数是按值传递的方式进行的。值传递过程中，被调函数的形式参数作为被调函数的局部变量处理，即在堆栈中开辟了内存空间以存放由主调函数放进来的实参的值，从而成为实参的一个副本。值传递的特点是被调函数对形式参数的任何操作都是作为局部变量进行，不会影响主调函数的实参变量的值。以按值传递方式进行参数传递的过程如下：

- ① 计算出实参表达式的值，接着给对应的形参变量分配一个存储空间，该空间的大小等于该形参类型的长度。
- ② 把已求出的实参表达式的值一一存入到为形参变量分配的存储空间中，成为形参变量的初值，供被调用函数执行时使用。

这种传递把实参表达式的值传送给对应的形参变量，故称这种传递方式为“按值传递”。这种方式被调用函数本身不对实参进行操作，也就是说，即使形参的值在函数中发生了变化，实参的值也完全不会受到影响，仍为调用前的值。

【范例 5-4】函数的传值调用。该范例定义了一个交换两个数的函数 swap，在主函数 main() 中采用传值调用该函数，读者可查看其输出，实现代码如代码清单 5-4 所示。

代码清单 5-4

```
1  #include <iostream.h>                //预处理文件
2  void swap(int a, int b);             //函数原型的声明
3  int main()                          //主函数
4  {
5      int x=8,y=10;                   //声明变量
6      cout<<"x="<<x<<"    y="<<y<<endl;    //输出调用函数前 x 和 y 的值
7      swap(x,y);                     //调用函数 swap
8      cout<<"x="<<x<<"    y="<<y<<endl;    //输出调用函数后 x 和 y 的值
9      return 0;
10 }
```

```

11 void swap(int a,int b)                //交换两个数的函数
12 {   int t;
13     t=a;
14     a=b;
15     b=t;                             //交换
16 }

```

【运行结果】在 Visual C++ 中执行上述程序，其结果如图 5-7 所示。

【范例解析】上述代码首先输出未调用 swap 函数前 x、y 的值，在调用 swap 函数后，再一次输出 x、y 的值，看其是否达到了交换的功能。由图 5-7 的运行结果读者可以看出，x、y 这两个值并没有交换，也即没有达到预期的目的，这就是传值调用。

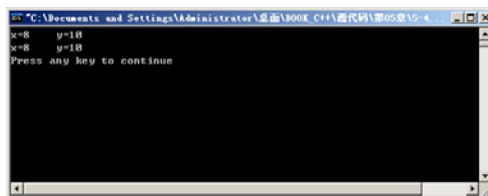


图 5-7 传值调用

简单来说，传值调用就是指当一个函数被调用时，C++ 根据实参和形参的对应关系将实参的值一一复制给形参，即实参的值单向传递给形参。函数本身不对实参进行任何操作，即使形参的值在函数中改变，实参的值也不会受到影响。



提示 为使程序可靠和便于调试，在程序中一般不改变实参的值，这时可采用按值传递的方式。

5.3.3 引用调用

函数的传值调用方式虽然容易理解，但形参值的改变不能对实参产生影响。因此传值调用方式在许多地方不适合用，如上述的两个数之间的交换函数，就无法用传值调用实现。此处就需要以引用作为参数，既能完成传值调用的功能，又使函数调用显得方便自然。

引用传递过程中，被调函数的形式参数虽然也作为局部变量在堆栈中开辟了内存空间，但是这时存放的是由主调函数放进来的实参变量的地址。被调函数对形参的任何操作都被处理成间接寻址，即通过堆栈中存放的地址访问主调函数中的实参变量。正因为如此，被调函数对形参做的任何操作都影响了主调函数中的实参变量。

引用传递方式是在函数定义时在形参前面加上引用运算符“&”。在函数被调用时，参数传递的内容不是实参的值，而是实参的地址，即将实参的地址放到 C++ 为形参分配的内存空间中，因此形参的任何操作都会改变相应实参的值。

【范例 5-5】函数的引用调用。要实现上述示例的在主函数 main() 中调用交换函数 swap，使得两个数之间完成交换，就可以使用引用调用来实现，代码如代码清单 5-5 所示。

代码清单 5-5

```

1  #include <iostream.h>                //预处理文件
2  void swap(int &a, int &b);           //函数原型的声明
3  int main()                          //主函数
4  {
5      int x=8,y=10;                   //声明变量
6      cout<<"x="<<x<<"    y="<<y<<endl;    //输出调用函数前 x 和 y 的值
7      swap(x,y);                     //调用函数 swap
8      cout<<"x="<<x<<"    y="<<y<<endl;    //输出调用函数后 x 和 y 的值
9      return 0;
10 }                                   //主函数结束
11 void swap(int &a,int &b)           //交换两个数的函数,形参前加上了&
12 {   int t;

```



```

13     t=a;
14     a=b;
15     b=t;                                //交换
16 }

```

【运行结果】上述代码的执行结果如图 5-8 所示。

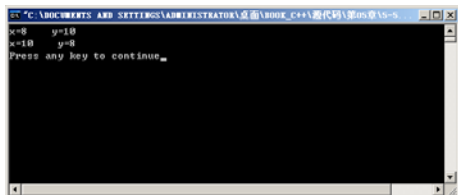


图 5-8 引用调用

【范例解析】读者应该注意到了，代码 5-5 与代码 5-4 的区别就在于定义函数 `swap` 时，其参数前加上了 `&` 符号，其余代码均一致。增加了 `&` 符号即表示该函数被调用时采用的是引用调用，传递给函数的是实参的地址，因此，能够实现交换的功能。



由于传递的是地址，在调用函数时不创建新的参数变量（开辟新的内存空间），因此在程序中对于占有内存较多的数据参数，为了节省内存，可采用引用传递的方式。

此外，引用传递还可以借助于指针（指针的概念将在后续章节中介绍），即在函数定义时，将形参说明成指针，而调用函数时就需要指定地址值形式的实参，这种参数传递方式也称地址传递，此处暂不作介绍。

5.3.4 嵌套调用

前面章节介绍到了，C++ 函数不能嵌套定义，即一个函数不能在另一个函数体中进行定义。但在使用时，允许函数的嵌套调用，即在调用一个函数的过程中又调用另一个函数，并且函数嵌套调用的层次可是多层的。例如，下面定义了一个函数 `func1`，而在 `func2` 的函数体中调用了 `func2`，这就是函数的嵌套定义，代码如下所示。

```

func1(int a, float b)
{
    float c;
    c=func2(b-1,b+1);
}
int func2(float x, float y)
{
    函数体
}

```



此处需要注意的是，`func1` 和 `func2` 是分别独立定义的函数，互不从属。并且在 `func1` 中调用 `func2` 函数前，`func2` 函数已经被声明。

5.3.5 递归调用

在函数的调用中，还有一个较为特殊的情况。比如一个函数直接或间接地调用自身，这种现象就是函数的递归调用。递归调用有两种方式：直接递归调用和间接递归调用。直接递归调用即在一个函数中调用自身，间接递归调用即在一个函数中调用了其他函数，而在该其他函数中又调用了本函数。

递归调用的执行包括两个步骤：递推和回归。利用函数的递归调用，可将一个复杂问题分解为一个相对简单且可直接求解的子问题（“递推”阶段）；然后将这个子问题的结果逐层进行

回代求值, 最终求得原来复杂问题的解 (“回归” 阶段)。

【范例 5-6】函数的递归调用。该范例求出了整数 n 的阶乘, 其采用的就是函数的递归调用, 实现代码如代码清单 5-6 所示。

代码清单 5-6

```

1  #include <iostream.h>
2  int Fac(int n)                                //定义函数
3  {
4      if(n<0)
5      {
6          cout<<"error!"<<endl;
7          return(-1);                            //返回错误结果
8      }
9      else if(n<=1)                             //整数 n<=1 成立
10         return(1);
11     else
12         return (n*f(n-1));                    //递归调用自身
13 }
14 void main()
15 {
16     int Fac(int n);                            //声明函数
17     int n;                                    //定义整型变量 n
18     cout<<"input n:"<<endl;
19     cin>>n;                                    //接收键盘输入
20     cout<<"n!="<<Fac(n)<<endl;                //调用函数
21 }
```

【运行结果】在 Visual C++ 中输入上述代码, 该程序的运行结果如图 5-9 所示。

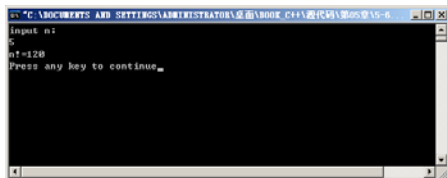


图 5-9 递归调用

【范例解析】读者可以看出, 在定义函数 $\text{Fac}()$ 的函数体中, 其调用了本身来完成计算任务, 这就是函数的递归调用。当 $n=5$ 时, 其执行流程为: 先递推, 其执行流程图如图 5-10 所示。



前面提到了, 递归就是先完成函数的递推, 再进行回归。当图 5-10 递推到 $\text{Fac}(1)$ 时, 递推完成, 开始回归, 如图 5-11 所示。

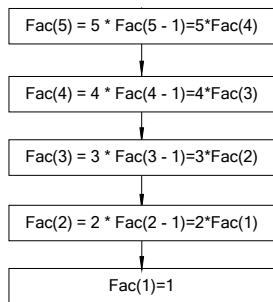


图 5-10 递推流程

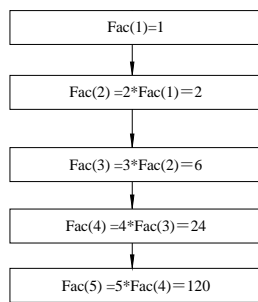


图 5-11 回归流程



简单来说，该程序的执行顺序如图 5-12 所示。

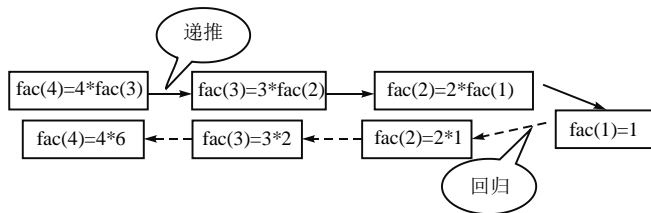


图 5-12 递归执行顺序

使用函数的递归调用读者需要注意以下三个方面。

- 递归算法设计简单，但消耗的上机时间和占据的内存空间比非递归大。
- 设计一个正确的递归过程或函数过程必须具备两点：具备递归条件；具备递归结束条件。
- 一般而言，递归函数过程对于计算阶乘、级数、指数运算有特殊效果。

5.3.6 带默认形参值的函数

在 C++ 语言中调用函数时，通常要为函数的每个形参给定对应的实参。若没有给出实参，则按指定的默认值进行工作。当一个函数既有定义又有声明时，形参的默认值必须在声明中指定，而不能在定义中指定。只有当函数没有声明时，才可以在函数定义中指定形参的默认值。此外，默认值的定义必须遵守从右到左的顺序，如果某个形参没有默认值，则它左边的参数就不能有默认值。例如：

```
void func1(int a, double b=4.5, int c=3);           //合法
void func1(int a=1, double b, int c=3);           //不合法
```

在进行函数调用时，实参与形参按从左到右的顺序进行匹配，当实参的数目少于形参时，如果对应位置形参又没有设定默认值，就会产生编译错误；如果设定了默认值，编译器将为那些没有对应实参的形参取默认值。



警告 形参的默认值可以为全局常量、全局变量、表达式、函数调用，但不能为局部变量。例如，下面的程序是不合法的：

```
void func1()
{
    int k;
    void g(int x=k); //k 为局部变量
}
```

5.4 变量的作用域

变量的作用域是指该变量的作用范围。在具体讲解作用域相关概念前，读者应对程序的内存区域有一些了解。一个程序将操作系统分配给其运行的内存块分为 4 个区域：

- 代码区，存放程序的代码，即程序中各个函数中的代码块。
- 全局数据区，存放程序全局数据和静态数据。
- 堆区，存放程序的动态数据。
- 栈区，存放程序的局部数据，即各个函数中的数据。

正因为变量要实现的存储功能不一样，因此其在内存块的区域也不同。在 C++ 中，根据变量的作用域可将变量划分为局部变量和全局变量。

5.4.1 局部变量

在一个函数内部说明的变量是内部变量，其只在该函数范围内有效。也就是说，只有在包含变量说明的函数内部，才能使用被说明的变量，在此函数之外就不能使用这些变量了。因此，这些内部变量被称为局部变量。

前面介绍的代码清单 5-4 中定义了一个函数 swap，该函数中声明了整型变量 t，该变量就是一个局部变量，其只在 swap 函数中起作用。

【范例 5-7】局部变量的应用。该范例对代码 5-4 做一些修改，读者仔细观看局部变量的作用范围，代码如代码清单 5-7 所示。

代码清单 5-7

```

1  #include <iostream.h>                                //预处理文件
2  void swap(int a, int b);                             //函数原型的声明
3  int main()                                           //主函数
4  {
5      int x=8,y=10;                                    //声明变量
6      int t=0;
7      cout<<"x="<<x<<"    y="<<y<<endl;            //输出调用函数前 x 和 y 的值
8      swap(x,y);                                       //调用函数 swap
9      cout<<"x="<<x<<"    y="<<y<<endl;            //输出调用函数后 x 和 y 的值
10     cout<<"t in main is : "<<t<<endl;
11     return 0;
12 }                                                     //主函数结束
13 void swap(int a,int b)                               //交换两个数的函数
14 {
15     int t;
16     t=a;
17     a=b;
18     b=t;                                             //交换
19     cout<<"t in swap is : "<<t<<endl;              //输出结果
20 }
```

【运行结果】上述程序在 Visual C++ 中的运行结果如图 5-13 所示。

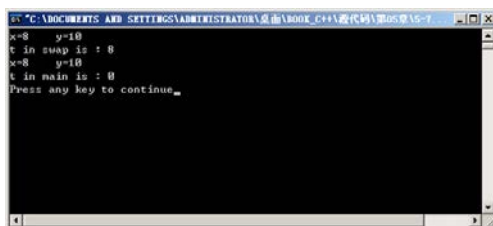


图 5-13 局部变量

【范例解析】范例 5-7 代码中，在 swap() 函数中定义了变量 t，用于交换两个数的中间变量，而在 main() 函数中也定义了一个变量 t，并给其赋初值 0。从上述程序读者可以看出，main() 函数和 swap() 函数中的变量 t 都是局部变量，其都有自身的作用范围。



注意 在 C++ 中，主函数 main() 中定义的内部变量，只能在主函数中使用，其他函数不能使用。同时，主函数中也不能使用其他函数中定义的内部变量。因为主函数也是一个函数，与其他函数是平行关系。

上述代码中，如果主函数 main() 要使用 swap() 函数中定义的变量 t，将 main() 函数中的变



量 `t` 的声明语句即第 6 行注释掉，这是不允许的，Visual C++ 编译程序将给出变量没有定义的错误，如图 5-14 所示。



图 5-14 错误信息

此外，在使用局部变量时需注意如下几个事项：

- 形参变量也是内部变量，属于被调用函数；实参变量，则是调用函数的内部变量。
- 允许在不同的函数中使用相同的变量名，但它们代表不同的对象，分配不同的单元，互不干扰，也不会发生混淆。
- 在复合语句中也可定义变量，其作用域只在复合语句范围内。

5.4.2 全局变量

与局部变量相对应，C++ 中也有全局变量的概念。全局变量又称为外部变量，其是在函数外部定义的变量。



提示 全局变量不属于任何一个函数，可被作用域内的所有函数直接引用，其作用域从外部变量的定义位置开始，到本文件结束为止。

【范例 5-8】全局变量的应用。该范例声明了 3 个全局变量 `s1`、`s2` 和 `s3`，用于接收用户输入的长方体的长 (`l`)、宽 (`w`)、高 (`h`)，求长方体体积及正、侧、顶三个面的面积，实现代码如代码清单 5-8 所示。

代码清单 5-8

```
1  #include <iostream.h>
2  int s1,s2,s3;                                //声明全局变量 s1、s2 和 s3
3  int vs(int a,int b,int c)                    //定义函数
4  {
5      int v;
6      v=a*b*c;                                //计算体积
7      s1=a*b;                                  //计算正面积
8      s2=b*c;                                  //计算侧面积
9      s3=a*c;                                  //计算顶面积
10     return v;                                //只返回体积变量
11 }
12 void main()
13 {
14     int v,l,w,h;                              //定义存储长、宽、高的变量
15     cout<<"Please input length,width and height:"<<endl;
16     cin>>l>>w>>h;                            //接收键盘输入
17     v=vs(l,w,h);                              //调用函数
18     cout<<"v= "<<v<<endl;
19     cout<<"s1= "<<s1<<endl;                    //直接引用全局变量
20     cout<<"s2= "<<s2<<endl;
21     cout<<"s3= "<<s3<<endl;
22 }
```

【运行结果】该程序在 Visual C++ 中的运行结果如图 5-15 所示。

【范例解析】上述代码中,程序一开始就声明了三个变量 s1、s2 和 s3,这三个都是全局变量,其作用域为该文件内所有函数。因此,在函数 vs() 中使用到了,而在主函数 main() 中同样使用到了。

读者可以看出,全局变量能够很方便地完成某些功能,能够简化程序。然而,这也使得函数之间的独立性降低了,这并不符合结构化程序设计的思想。因此,在使用全局变量时,如果可用其他方法实现,则尽量少用全局变量实现。

此外,对于全局变量还有以下几点说明:

- 外部变量可加强函数模块之间的数据联系,但又使这些函数依赖这些外部变量,从而使得这些函数的独立性降低。
- 在同一源文件中,允许外部变量和内部变量同名。在内部变量的作用域内,外部变量将被屏蔽而不起作用。
- 全局变量的作用域是从定义点到本文件结束。如果定义点之前的函数需要引用这些外部变量时,需要在函数内对被引用的外部变量进行说明。

外部变量说明的一般形式为:

```
extern 数据类型 外部变量1,外部变量2...;
```

提示

全局变量的定义和说明是不一样的。全局变量的定义,其必须在所有的函数之外定义,且只能定义一次。而外部变量的说明,一般出现在要使用该外部变量的函数内,而且可以出现多次。

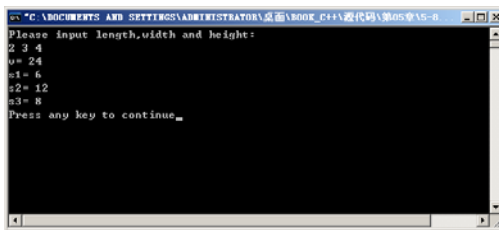


图 5-15 全局变量

5.5 函数的作用域

每个函数都构成了一个函数作用域,函数作用域的概念跟变量的存储位置 and 生命期有关。函数的参数和在函数中声明并定义的变量即局部变量,其被分配在堆栈上,随着函数的执行而生成,随着函数的退出而消亡。

标号是唯一具有函数作用域的标识符,goto 语句使用标号。标号声明使得该标识符在一个函数内的任何位置均可以被使用。

【范例 5-9】标号的声明和 goto 语句的使用。声明了两个标号,并在函数中调用 goto 语句使用标号,代码如代码清单 5-9 所示。

代码清单 5-9

```
1  #include <iostream.h>
2  void fn()                                //定义函数
3  {
4      goto S;                              //跳转到标号 s
5      int b;
6      cin>>b;                              //接收键盘输入
7      if(b>0)
8      {
9          S:                                //定义标号
10         goto End;                         //跳转到 End
11     }
```



```

12 End:                                     //定义标号
13     cout<<"Can't input b"<<endl;
14 }
15 void main()                             //主函数
16 {
17     fn();                               //调用函数
18 }

```

【运行结果】上述代码在 Visual C++ 中运行，其运行结果如图 5-16 所示。

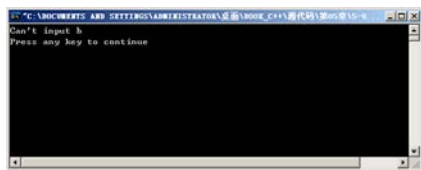


图 5-16 函数作用域

【范例解析】上述代码使用了 goto 语句，其中的 S 和 End 均为标号。代码中 goto 语句使得其下的变量定义和输入等语句没有被执行，不能接收用户输入并判断，而是直接跳转到了输出“Can't input b”的信息。

因此，goto 或 switch 语句不应使控制在一个声明的作用域之外跳到该声明的作用域内，因为这种跳转越过了变量的声明语句，使变量不能被初始化。



注意 在进行函数原型声明时所作的参数声明表示参数在该函数作用域中，该作用域开始于函数原型声明的左括号，结束于函数原型声明的右括号。

例如，下面的代码是函数 Area() 的原型声明：

```
void Area(double width, double length)
```

其中的参数声明(double width,double length)只在圆括号之内有效，在程序的其他地方使用 width 和 length 必须另外有定义，否则会引起变量未定义的编译错误。例如，下面的代码引起一个无定义的标识符编译错误：

```

void Area(double width, double length);
length=50;                                     //error length 无定义

```

所以，在这个函数原型声明中的标识符 width 和 length 是可有可无的。即上面的函数原型等价于下面的函数原型声明：

```
void Area(double, double);
```

但是，参数中有了标识符，可以增强可读性。上面参数中带标识符的函数原型声明使人一看就明白一个参数是宽度值，另一个是长度值。所以，习惯上，在函数原型声明中，都为参数指定一个有说明意义的标识符，而且一般总是与该函数定义中参数的标识符一致。即：

```

void fn(int number);                           //函数声明
//...
void fn(int number)                             //函数定义
{
    //...
}

```

5.6 函数重载

函数重载是指同一个函数名可以对应着多个函数的实现。每一类实现对应着一个函数体，这些函数的名字相同，但是函数的参数的类型不同，这就是函数重载。

例如，给同一个名为 sum() 的函数定义两个不同的函数体，该函数的功能是求两个操作数的和。其中，一个函数实现求两个整数之和，另一个则求两个浮点型数之和。而这两种功能都可以通过调用同一个名为的 sum() 函数来实现。

5.6.1 函数的重载

函数重载又称为函数的多态性,是指同一个函数名对应着多个不同的函数。所谓“不同”,是指这些函数的形参表必须互不相同,或者是形参的个数不同,或者是形参的类型不同,或者是两者都不相同,否则将无法实现函数重载。例如,下面是合法的重载函数:

```
int func1(int,int);
int func1(int);
double func1(int,long);
double func1(long);
```

重载函数的类型,即函数的返回类型,可以相同,也可以不同。但如果仅仅是返回类型不同而函数名相同、形参表也相同,则是不合法的,编译器会给出语法错误提示。例如:

```
int func1(int a, int b);
double func1(int a, int b);
```



警告 除形参名外都相同的情况,编译器不认为是重载函数,只认为是对同一个函数原型的多次声明,因此上述声明方法是错误的。

在调用一个重载函数 `func1()` 时,编译器必须判断函数名 `func1` 到底是指哪个函数。例如,Visual C++ 中通过编译器,根据实参的个数和类型对所有 `func1()` 函数的形参一一进行比较,从而调用一个最匹配的函数。

5.6.2 参数类型不同的函数重载

前面提到了,函数要进行重载,其必须是函数的参数类型不同或个数不同,下面分别介绍这两种不同方式的函数重载的实现。

【范例 5-10】 参数类型不同的函数重载。该范例给出了两个参数类型不同的函数 `add`,其分别实现两个整数的相加和两个浮点数的相加,实现代码如代码清单 5-10 所示。

代码清单 5-10

```
1  #include <iostream.h>
2  int add(int, int);           //声明计算整型数值的函数 add
3  double add(double, double); //声明计算浮点型数值的函数 add
4  void main()
5  {
6      cout<<add(1,2)<<endl;    //调用第一个 add 函数
7      cout<<add(1.2,2.2)<<endl; //调用第二个 add 函数
8  }
9  int add(int x, int y)       //定义计算整型数值的函数 add
10 {
11     return x+y;
12 }
13 double add(double a, double b) //定义计算浮点型数值的函数 add
14 {
15     return a+b;             //返回计算结果
16 }
```

【运行结果】 该程序的运行结果如图 5-17 所示。

【范例解析】 该程序中, `main()` 函数中调用相同名字 `add` 的两个函数,前边一个 `add()` 函数对应的是两个整型 `int` 数求和的函数实现,而后边一个 `add()` 函数对应的是两个浮点型 `double` 数求和的函数实现,这便是函数的重载。

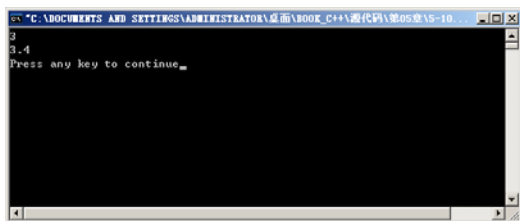


图 5-17 参数类型不同的函数重载



提示 读者可以看出，在参数类型不同的情况下，系统能自动分辨在调用时应使用哪个函数，从而得出正确的结果。

5.6.3 参数个数上不同的重载函数

前面提到了，除了 5.6.2 节介绍的参数类型不同的函数可以进行重载外，针对参数个数不同的函数也可以进行重载。

【范例 5-11】参数个数上不同的重载函数。该范例定义了 3 个函数，其函数名都为 min，其参数个数分别为 2 个、3 个和 4 个，其功能分别为取 2 个、3 个和 4 个整型数据的最小值，其实现代码如代码清单 5-11 所示。

代码清单 5-11

```

1  #include <iostream.h>
2  int min(int a, int b);           //声明带有 2 个参数的函数 min
3  int min(int a, int b, int c);    //声明带有 3 个参数的函数 min
4  int min(int a, int b, int c, int d); //声明带有 4 个参数的函数 min
5  void main()
6  {
7      cout<<min(2,3)<<endl;        //调用第一个函数 min
8      cout<<min(3,4,5)<<endl;      //调用第二个函数 min
9      cout<<min(4,5,6,7)<<endl;    //调用第三个函数 min
10 }
11 int min(int a, int b)             //定义带有 2 个参数的函数 min
12 {
13     return a;
14 }
15 int min(int a, int b, int c)      //定义带有 3 个参数的函数 min
16 {
17     int t = min(a, b);            //调用函数
18     return min(t,c);              //返回调用函数的值
19 }
20 int min(int a, int b, int c, int d) //定义带有 4 个参数的函数 min
21 {
22     int t1 = min(a, b);           //调用函数
23     int t2 = min(c, d);           //调用函数
24     return min(t1, t2);          //返回调用函数的值
25 }

```

【运行结果】在 Visual C++ 中的运行结果如图 5-18 所示。

【范例解析】该程序中出现了函数重载，函数名 min 对应有三个不同的实现，函数的区分依据参数个数不同，这里的三个函数实现中，参数个数分别为 2、3 和 4，在调用函数时根据实参的个数来选取不同的函数实现。

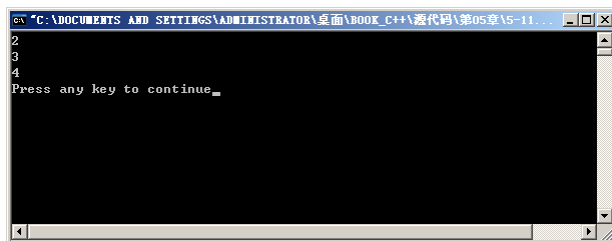


图 5-18 参数个数上不同的重载函数

读者可以看出, 在函数调用时, 程序自动根据实际参数的个数来选取不同的 `min()` 函数, 从而得到正确的结果。



提示 事实上, 函数重载在类和对象应用比较多, 尤其是在类的多态性中。

5.7 小结

本章主要介绍了 C++ 中函数的相关内容, 函数是程序的组成部分, 鉴于其重要性, 甚至可以说程序就是由一个个的函数组成的。本章详细讲解的内容主要包括函数的定义、声明和调用, 这是在实际程序中使用较多的。此外, 为了方便读者更好地理解函数的定义和调用原理, 本章还就变量和函数的作用域做了简单的介绍。同时, 由于 C++ 是一种面向对象的程序设计语言, 因此本章还详细讲解了函数的重载概念及其两种方式的函数重载。

5.8 习题

1. 编写函数, 重复打印给定字符 `n` 次。例如, 在主函数中调用该函数后, 给出打印字符和次数, 该函数将在屏幕上输出指定字符。

【解答】该习题主要考查字符串参数在函数调用过程中的传递。根据前面章节的学习, 读者知道字符的输出可以通过输出流 `cout` 来实现, 但更好的办法是通过 `putchar` 函数来实现。指定输出字符的次数后, 通过一个循环语句来实现即可。同时, 在主函数中要指定需打印的字符, 也可以用函数 `getchar` 来实现接收。其简要实现代码如下所示。

```
void printchar(char, int);

printchar(c, n);

void printchar(char ch, int count)          /* 重复打印字符 ch, count 表示次数 */
{
    int i;
    for(i=1; i<=count; i++)
        putchar(ch);
}
```

2. 下列程序的输出结果是多少?

```
#include <iostream.h>
int fun(int x, int y)
{ return x*y; }
void main()
{
    int k=5;
    cout<<fun(k++, ++k)<<endl;
}
```



【解答】该习题主要考查函数的调用。上述程序中的函数 fun 包含两个参数，在调用时将实参传递给形参。读者需要注意实参分别为 k++和++k，进行参数传递后，k 的值都将加 1。因此，传递到函数 fun 后，x 和 y 都变为 6，运行函数后，得到函数返回值 36。因此，该程序的输出为 36。

3. 已知三角形的三边，求三角形面积，将其编写成一个函数，在 main 函数中调用该函数。例如，运行该程序段后输入三角形的三边分别为 3、4、5，输出三角形面积如图 5-19 所示。

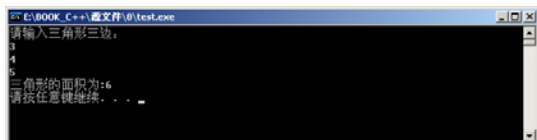


图 5-19 求三角形的面积

【解答】该习题主要考查函数的声明、定义和调用。在使用函数前需要先对其进行声明，声明后的函数定义可写在调用后。该函数求三角形面积，因此函数的参数为三边，根据三角形面积的计算公式：面积=三边之和除 2 再乘以其与三边之差的平方根。需要读者注意，此处的函数返回值应为浮点型，因为三角形面积有可能为浮点数。

4. 输入的整数按字符串形式逆序输出，要求分别用递归算法和非递归算法实现。如：输入 12345，输出：5 4 3 2 1。

【解答】该程序段首先需声明两个函数，其分别对应非递归输出分解的结果和递归输出，在主函数 main()中调用这两个函数。其中，非递归的函数实现中可使用 while 循环，其循环变量 a 以 a/10 为依次一次循环。其简要实现代码如下所示。

```
#include<iostream.h>                                //包含头文件
void print(int a);                                   //声明函数 print
void print1(int a);                                  //声明函数 print1
void main()
{
    int n;                                           //定义整型变量 n
    cin>>n;                                         //接收用户的键盘输入
    cout<<"非递归输出: ";                          //输出提示
    print(n);                                       //调用非递归函数 print
    cout<<endl;                                    //输出换行
    cout<<"递归输出: ";                            //输出提示
    print1(n);                                     //调用递归函数 print1
    cout<<endl;
}
void print(int a)                                    //定义非递归函数
{
    while(a>0)                                     //a>0 成立则一直执行
    {
        cout<<a%10<<" ";                          //循环输出位
        a=a/10;                                    //a 的值为 a 整除 10 后的商
    }
}
void print1(int a)                                   //定义递归函数
{
    if(a>0)                                         //a>0 成立则执行
    {
        cout<<a%10<<" ";                          //循环输出位
        print1(a/10);                              //递归输出
    }
}
```

}

5. 下面的程序段的运行结果是多少?

```
#include <iostream.h>
int func(int x)
{
    int p;
    if (x==0 || x==1) return (3);
    p=x- func(x-2);
    return p;
}
void main( )
{
    cout<<func(9);
}
```

【解答】该习题主要考查递归函数及其返回值。递归函数是在定义中调用自身的函数，上述程序中在定义时调用了其自身。要计算递归函数的返回值，需要将实参传递到函数中后，先进行递推，再进行回归来实现。上述程序中实参为 9，那么第一次递推后 p 的值为 $(9-\text{func}(7))$ ，第二次递推后 p 的值为 $(7-\text{func}(5))$ ，一直到 x 的值为 0 或为 1 后再回归，得出最终 p 的值为 7，如图 5-20 所示，因此该程序的返回值为 7。

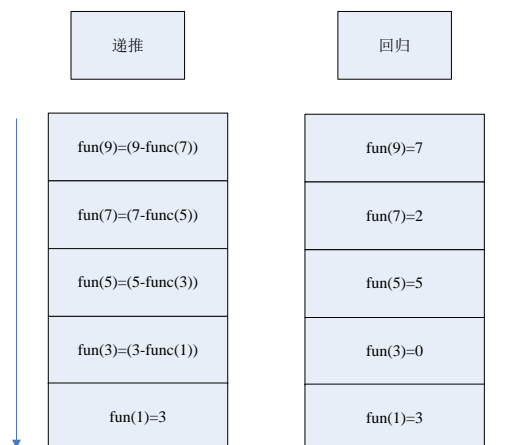


图 5-20 递推与回归

第 6 章 编译预处理

预处理是 C++ 的一个重要功能，其是指编译器在进行第一遍扫描之前所做的工作，其由预处理程序负责完成。C++ 编译器对一个源文件进行编译时，它将自动调用预处理程序对源程序中的预处理部分作处理，处理完成后才进行编译。

C++ 提供的预处理功能，包括宏定义、包含文件处理、条件编译等。本章将要介绍的是常用的几种预处理功能。以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为学习的参照。

- 了解预处理命令的功能。
- 掌握宏定义及其使用。
- 掌握文件包含的使用。
- 掌握常用的编译预处理命令。

6.1 预处理命令

简单来说，预处理就是对源文件进行编译前，先对预处理部分进行处理，然后对处理后的代码进行编译。这样做的好处是，经过处理后的代码，将会变得很精短。为用户更好地使用预处理，C++ 提供了丰富的预处理命令，主要包括如下几种：`#define`、`#error`、`#if`、`#else`、`#elif`、`#endif`、`#ifdef`、`#ifndef`、`#undef`、`#line` 和 `#pragma`。

由上述命令读者可以看出，每个预处理指令均带有符号“#”。简单来说，上面的这些预处理命令可以划分为文件包含、条件编译、布局控制和宏替换 4 个大类：

- 文件包含（`#include`）是一种最为常见的预处理，主要是作为文件的引用组合源程序正文。
- 条件编译（`#if`、`#ifndef`、`#ifdef`、`#endif`、`#undef` 等）也是比较常见的预处理，主要是进行编译时进行有选择的挑选，注释掉一些指定的代码，以达到版本控制、防止对文件重复包含的功能。
- 布局控制（`#pragma`）也是应用预处理的一个重要方面，主要功能是为编译程序提供非常规的控制流信息。
- 宏替换（`#define`）是最常见的用法，其可以定义符号常量、函数功能、重新命名、字符串的拼接等各种功能。

6.2 宏

宏（macro）是程序设计语言中使用较为广泛的一个概念，简单来说，宏是一种以相同的源语言执行预定义指令序列的指令。在 C++ 中，通过宏的使用，可以将一个表达式定义成宏，并在 C++ 的源程序中随意调用。

6.2.1 宏概述

在 C++ 语言源程序中允许用一个标识符来表示一个字符串，称为“宏”。被定义为宏的标识符称为宏名。在编译预处理时，对程序中所有出现的宏名，都用宏定义中的字符串去代换，这称为宏代换或宏展开。在定义宏时要注意如下的事项：

- 宏名一般用大写。
- 使用宏可提高程序的通用性和易读性，减少不一致性，减少输入错误和便于修改。例如，数组大小常用宏定义。
- 预处理是在编译之前的处理，而编译工作的任务之一就是语法检查，预处理不作语法检查。



提示 宏定义是由源程序中的宏定义命令完成的，宏代换是由预处理程序自动完成的。在 C++ 中，宏分为有参数和无参数两种。

6.2.2 不带参数的宏定义

不带参数的宏也称为无参宏，其宏名后不带参数，定义的一般形式为：

#define 标识符 字符串

其中，参数定义如下：

- “#”表示这是一条预处理命令。凡是以“#”开头的均为预处理命令，“#define”为宏定义命令。
- “标识符”为所定义的宏名。
- “字符串”可以是常数、表达式、格式串等。在前面介绍过的符号常量的定义就是一种不带参数的宏定义。

【范例 6-1】不带参数的宏定义的使用。该范例中的宏定义#define M(y*y+3*y)定义了 M 表达式 (y*y+3*y)，在编写源程序时，所有的 (y*y+3*y) 都可由 M 代替，而对源程序作编译时，将先由预处理程序进行宏代换，即用 (y*y+3*y) 表达式去置换所有的宏名 M，然后再进行编译，代码如代码清单 6-1 所示。

代码清单 6-1

```

1  #define M (y*y+3*y)                //定义不带参数的宏
2  #include <iostream.h>
3  void main()
4  {
5      int s,y;                        //定义变量
6      cout<<"Input a number:";
7      cin>>y;                         //接收输入
8      s=3*M+4*M+5*M;                 //使用宏
9      cout<<s<<endl;                 //输出
10 }
```

【运行结果】上述代码在 Visual C++ 中运行，其结果如图 6-1 所示。

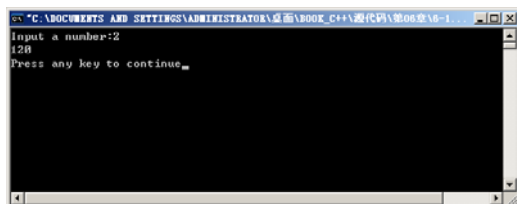


图 6-1 宏定义

【范例解析】上例程序中首先进行宏定义，定义 M 表达式 (y*y+3*y)，在 s=3*M+4*M+5*M 中作了宏调用。在预处理时经宏展开后该语句变为：



```
s=3*(y*y+3*y)+4(y*y+3*y)+5(y*y+3*y);
```



警告 在宏定义中，表达式(y*y+3*y)两边的括号不能少。否则会发生错误。例如上述代码中，将宏定义改为以下定义：

```
#define M y*y+3*y
```

在宏展开时将得到下述语句：

```
s=3*y*y+3*y+4*y*y+3*y+5*y*y+3*y;
```

这显然与原题意要求不符，计算结果当然也是错误的。因此在进行宏定义时必须十分注意。应保证在宏代换之后不发生错误。对于宏定义，读者应注意以下几点：

- 宏定义是用宏名来表示一个字符串，在宏展开时又以该字符串取代宏名，这只是一种简单的代换，字符串中可以含任何字符，可以是常数，也可以是表达式，预处理程序对它不作任何检查。如有错误，只能在编译已被宏展开后的源程序中发现。
- 宏定义不是说明或语句，在行末不必加分号，如加上分号则连分号也一起置换。
- 宏定义必须写在函数之外，其作用域为从宏定义命令开始直到源程序结束。
- 一般来说，宏名用大写字母表示，以便于与变量区别，但也允许用小写字母。

6.2.3 取消宏

由于宏定义的作用域是整个源程序，在一些应用中不需要其覆盖整个程序，因此就需要终止其作用域，C++中终止其作用域的命令为`# undef`。如果要求宏定义只在一个函数中起作用，就可以在函数定义之前定义宏，在函数结束后结束宏。

【范例 6-2】取消宏。该范例中定义的宏只在 `main()` 函数中起作用，而在两个函数 `sout` 和 `lout` 中则无效，代码如代码清单 6-2 所示。

代码清单 6-2

```
1  # define PI 3.14159                                //定义宏
2  # include<iostream.h>
3  void sout(double);                                  //声明函数
4  void lout(double);                                  //声明函数
5  void main()
6  {
7      double radius,s,l;                               //定义变量
8      cout<<"Please input radius:";
9      cin>>radius;
10     s=PI*radius*radius;                               //使用宏
11     sout(s);                                           //调用函数
12     l=2*PI*radius;
13     lout(l);                                           //调用函数
14 }
15 # undef PI                                           //取消宏
16 void sout(double r)                                   //定义函数
17 {
18     r=PI*r/PI;
19     cout<<"the area is :"<<r<<endl;
20 }
21 void lout(double r)                                   //定义函数
22 {
23     cout<<"the long is :"<<r<<endl;
24 }
```

【运行结果】读者在 Visual C++ 中运行上述程序后, 系统会出现一个错误: 提示在函数 `sout` 中没有定义 `PI`, 结果如图 6-2 所示。



图 6-2 错误提示

而如果将上述代码中第 15 行终止宏定义的代码注释掉, 那么该程序可以被顺利执行, 完成计算圆面积和周长的功能, 运行结果如图 6-3 所示。

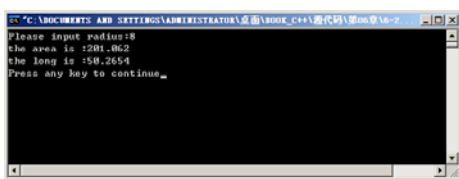


图 6-3 正确执行宏

【范例解析】读者从上述执行结果可以看出, 当使用 `# undef PI` 取消了宏后, 其在程序中就不再起作用了, 如果此时再调用宏, 编译系统将给出变量未定义的错误。



注意 以上介绍了宏定义的作用域及取消宏, 此外, 如果宏名在源程序中用引号括起来了, 那么预处理程序将不对其进行宏代换。

【范例 6-3】定义的宏是否被代换。该范例定义了宏 `OK`, 但预处理程序并没有在执行时进行宏代换并输出, 实现代码如代码清单 6-3 所示。

代码清单 6-3

```

1  #define OK 100                                //定义宏
2  #include <iostream.h>
3  void main()
4  {
5      cout<<"OK";                                //输出 OK, 并没有输出宏 OK 的内容
6      cout<<endl;                                //输出换行
7  }
```

【运行结果】在 Visual C++ 中运行上述程序, 运行结果如图 6-4 所示。

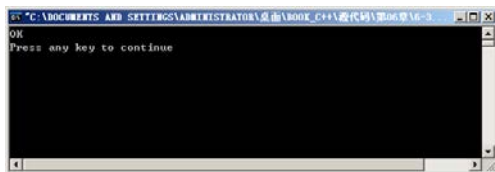


图 6-4 加引号的宏

【范例解析】上述代码定义了宏 `OK`, 并在 `main()` 函数中使用了 `OK`, 但因为其被双引号括起来了, 使预处理程序并没有对该宏进行代换, 因此其输出也并不是宏定义的值 100, 而是字符串 `OK`。



注意 凡是被双引号括起来的字符, 系统都不会进行宏代换, 而是直接输出其中的字符, 不进行任何改变。



6.2.4 宏嵌套

在宏定义中，读者还需要注意的是，宏定义允许嵌套，即在宏定义的字符串中可以使用已经定义的宏名，在宏展开时由预处理程序层层代换。

【范例 6-4】宏嵌套的实现。该范例定义了两个宏：PI 和 S，其中宏 S 的定义中使用到了宏 PI，实现代码如代码清单 6-4 所示。

代码清单 6-4

```
1  #define PI 3.1415926                //定义宏 PI
2  #define S PI*y*y                    //PI 是已定义的宏名
3  #include <iostream.h>
4  void main()
5  {
6      int y;                          //定义变量
7      cout<<"Please input the radius:"<<endl;
8      cin>>y;                          //接收键盘输入
9      cout<<S<<endl;                  //使用宏 s
10 }
```

【运行结果】上述代码分别定义了两个宏，在第二个宏 S 的定义中使用了第一个宏 PI，读者可以看到其运行结果如图 6-5 所示。

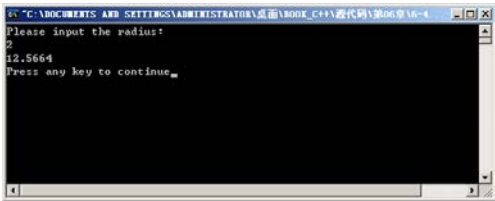



图 6-5 宏的嵌套定义

【范例解析】读者可以看出，在宏定义语句#define S PI*y*y 中，由于 PI 是宏名，预处理程序自动展开该宏，因此#define S PI*y*y 语句相当于如下语句：

```
#define S 3.1415926*y*y
```

6.2.5 带参数的宏定义

6.2.2 节介绍了不带参数的宏定义，这是常见的一种宏定义。事实上，C++中还允许宏带有参数。与函数定义时所带的参数一样，在宏定义中的参数称为形式参数，在宏调用中的参数称为实际参数。



提示 与不带参数的宏不同的是，带参数的宏在调用中，预处理程序不仅要展开宏，进行宏替换，而且要用实参去代换形参。

带参数的宏定义的一般形式为：

```
#define 宏名(形参表) 字符串
```

在字符串中含有各个形参。带参数的宏调用的一般形式为：

宏名(实参表)；

例如，下面定义了一个 M(y)的宏，其中 y 为该宏的参数，即形式参数，在调用该宏时用实际参数替换形式参数。

```
#define M(y) y*y+3*y                //宏定义
```

```
k=M(5);
```

在上述宏调用语句 `k=M(5);` 中的执行过程中, 实参 5 代替了形参 `y`, 经预处理宏展开后的语句为: `k=5*5+3*5`。

【范例 6-5】带参数的宏定义。该范例定义了一个带有参数的宏, 并在主程序中调用该宏, 读者应仔细理解该程序, 掌握带参数的宏的使用, 实现代码如代码清单 6-5 所示。

代码清单 6-5

```

1  #define MAX(a,b) (a>b)?a:b           //定义带参数的宏
2  #include <iostream.h>
3  void main()
4  {
5      int x,y,max;                     //定义两个整型变量
6      cout<<"input two numbers: ";
7      cin>>x>>y;                       //接收键盘输入
8      max=MAX(x,y);                   //使用宏 MAX(a,b)
9      cout<<"max="<<max<<endl;        //输出结果
10 }
```

【运行结果】在 Visual C++ 中运行上述程序, 其结果如图 6-6 所示。

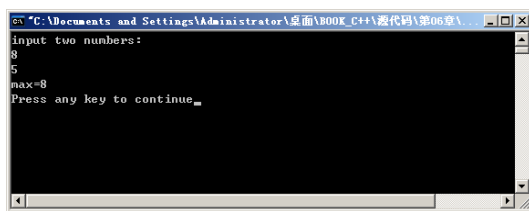


图 6-6 带参数的宏定义

【范例解析】上述程序的第一行进行带参宏定义, 用宏名 `MAX` 表示条件表达式 `(a>b)?a:b`, 形参 `a`、`b` 均出现在条件表达式中。程序第 8 行 `max=MAX(x,y)` 为宏调用, 实参 `x`、`y` 将代换形参 `a`、`b`, 而 `x`、`y` 来源于用户的输入。因此, 宏展开后该语句为:

```
max=(x>y)?x:y;
```

通过前面的内容读者已经知道了, 该语句使用了条件运算符, 其功能是用于计算 `x`、`y` 中的较大数。这就实现了带参数的宏在应用程序中的作用。

对于带参的宏定义, 读者应注意如下的几个事项。

- 带参宏定义中, 宏名和形参表之间不能有空格出现。例如把: `define MAX(a,b) (a>b)?a:b` 写为: `#define MAX (a,b) (a>b)?a:b` 将被认为是无参宏定义, 宏名 `MAX` 代表字符串 `(a,b)(a>b)?a:b`。宏展开时, 宏调用语句: `max=MAX(x,y);` 将变为: `max=(a,b)(a>b)?a:b(x,y);`, 这显然是错误的。
- 在带参宏定义中, 形式参数不分配内存单元, 因此不必作类型定义。而宏调用中的实参有具体的值。要用它们去代换形参, 因此必须作类型说明, 这是与函数中的情况不同的。在函数中, 形参和实参是两个不同的量, 各有自己的作用域, 调用时要把实参值赋予形参, 进行值传递。而在带参宏中, 只是符号代换, 不存在值传递的问题。



注意 在宏定义中的形式参数是标识符或者是变量, 而宏调用中的实参则不必一定是变量或常量, 也可以是表达式。

【范例 6-6】宏调用中实参为表达式。该范例定义了一个求平方的宏 `SQ`, 在主函数中调



用该宏时，实参是表达式 $a+1$ ，这在 C++ 中是允许的。实现代码如代码清单 6-6 所示。

代码清单 6-6

```

1  #define SQ(y) (y)*(y)                //定义带参数的宏 sq
2  #include <iostream.h>
3  void main()
4  {
5      int a,sq;                        //定义变量
6      cout<<"input a number: ";
7      cin>>a;                          //接收键盘输入
8      sq=SQ(a+1);                     //调用宏 sq
9      cout<<"sq="<<sq<<endl;         //输出结果
10 }
```

【运行结果】上述代码实现的功能是求出用户输入的数值加 1 后得出的平方，因此在实参中使用了表达式 $a+1$ ，它仅仅是一个变量，执行结果如图 6-7 所示。

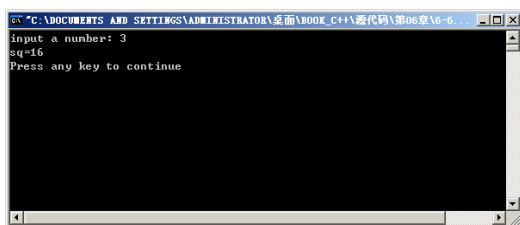


图 6-7 实参为表达式的宏调用

【范例解析】上述示例中，第 1 行为宏定义，形参为 y 。程序第 8 行宏调用中实参为 $a+1$ ，是一个表达式，在宏展开时，用 $a+1$ 替换 y ，再用 $(y)*(y)$ 替换 SQ ，得到如下语句：

```
sq=(a+1)*(a+1);
```



这与函数的调用是不同的，函数调用时要把实参表达式的值求出来再赋予形参。而宏代换中对实参表达式不作计算直接地代换。

此外，在宏定义中，字符串内的形参通常要用括号括起来以避免出错。在上例中的宏定义中 $(y)*(y)$ 表达式的 y 都用括号括起来，因此结果是正确的。

【范例 6-7】去除括号的宏定义。该范例将上述范例中定义的宏去掉括号，把程序改为如代码清单 6-7 所示，其执行结果有所不同。

代码清单 6-7

```

1  #define SQ(y) y*y                    //定义带参数的宏 sq
2  #include <iostream.h>
3  void main()
4  {
5      int a,sq;                        //定义变量
6      cout<<"input a number: ";
7      cin>>a;                          //接收键盘输入
8      sq=SQ(a+1);                     //调用宏 sq
9      cout<<"sq="<<sq<<endl;         //输出结果
10 }
```

【运行结果】读者可以比较代码 6-6 与代码 6-7，其唯一的不同在于第 1 行代码定义宏时没有将形参 y 用括号括起来，其他代码都一样，但其在 Visual C++ 的执行结果如图 6-8 所示。

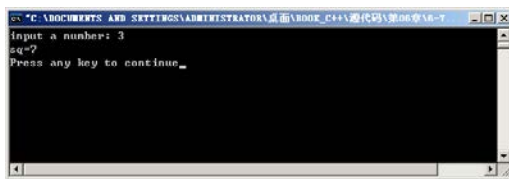


图 6-8 形参未加括号

【范例解析】读者可以看出，同样输入 3，但结果却是不一样的。这是由于代换只作符号代换而不作其他处理而造成的。上述示例中，宏代换后将得到以下语句：

```
sq=a+1*a+1;
```

由于 a 为 3，所以 sq 的值为 7，这显然与题意相违。因此，参数两边的括号是不能少的，即使参数只有一个变量。然而，有的时候即使在参数两边加括号还是不够的。

【范例 6-8】给宏参数加上括号的宏定义。该范例此处再将代码 6-6 做一些修改，给宏的定义中加上括号，代码如代码清单 6-8 所示。

代码清单 6-8

```

1  #define SQ(y) (y)*(y)                //定义宏 SQ，其变量加上了括号
2  #include <iostream.h>
3  void main()
4  {
5      int a,sq;                        //定义变量
6      cout<<"input a number: ";
7      cin>>a;                          //接收键盘输入
8      sq=16/SQ(a+1);                  //调用宏
9      cout<<"sq="<<sq<<endl;          //输出结果
10 }
```

【运行结果】读者可以看到，上述代码与代码 6-6 相比，只把宏调用语句改为：sq=16/SQ(a+1);，运行本程序如输入值仍为 3 时，希望结果为 1。但实际运行的结果如图 6-9 所示。

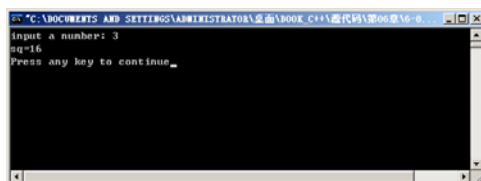


图 6-9 形参未加括号

【范例解析】此处仔细分析该宏定义，在宏代换之后调用语句变为：

```
sq=16/(a+1)*(a+1);
```

当用户输入 3，即 a 为 3 时，由于除号“/”和乘号“*”的运算符优先级和结合性相同，则先作运算 16/(3+1)得 4，再作 4*(3+1)最后得 16。为了得到正确答案，应在宏定义中的整个字符串外加括号。

【范例 6-9】给宏整体加上括号的宏定义。该范例在对程序进行修改，给宏定义的整体加上括号，代码如代码清单 6-9 所示。

代码清单 6-9

```

1  #define SQ(y) ((y)*(y))              //定义宏 SQ，其给整体都加上了括号
2  #include <iostream.h>
3  void main()
4  {
```



```

5      int a,sq;                                //定义变量
6      cout<<"input a number: ";
7      cin>>a;                                  //接收键盘输入
8      sq=16/SQ(a+1);                           //调用宏
9      cout<<"sq="<<sq<<endl;                 //输出结果
10     }

```

【运行结果】读者可以看到，代码 6-9 与代码 6-8 的区别就在于宏定义时对其形参的整体都增加了括号，其运行结果如图 6-10 所示。

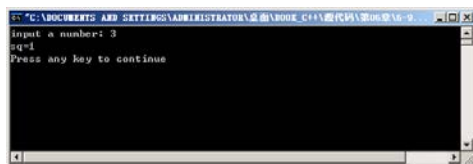


图 6-10 形参使用括号

【范例解析】从上述示例中读者可以看出，同样输入 3，即 a 为 3 时，得出的结果是正确的。再来看一下该调用语句的执行，由于宏定义语句的改变，在宏代换之后调用语句变为：

```
16/SQ((3+1)*(3+1))
```

因此，此处得出的结果是符合要求的。



注意 根据上述带参数的宏定义的不同方式，读者在进行带参数的宏定义时，需要时时关注宏展开后是否符合用户要求。

6.2.6 内联函数

内联函数也称为内嵌函数，当在一个函数的定义或声明前加上关键字 `inline` 则就把该函数定义为内联函数，它主要用于解决程序的运行效率。

计算机在执行一般函数的调用时，无论该函数多么简单或复杂，都要经过参数传递、执行函数体和返回等操作，这些操作都需要一定的时间。若把一个函数定义为内联函数后，在程序编译阶段，编译器就会在每次调用该函数的地方都直接替换为该函数体中的代码，由此省去函数的调用、相应地保存现场、参数传递和返回操作等所需的时间，从而加快整个程序的执行速度。

【范例 6-10】内联函数的应用。该范例定义了一个内联函数 `abs()`，其用于求一个整数的绝对值，再在主函数中调用该内联函数，实现代码如代码清单 6-10 所示。

代码清单 6-10

```

1  #include <iostream.h>
2  inline int abs(int x)                //定义内联函数 abs
3  {
4      return x<0?-x:x;                //输出一个整数的绝对值
5  }
6  void main()
7  {
8      int a,b=3,c,d=-4;                //定义变量并初始化
9      a=abs(b);                        //调用函数
10     c=abs(d);                        //调用函数
11     cout<<"a="<<a<<" ,c="<<c<<endl; //输出值
12 }

```

【运行结果】在 Visual C++ 中执行上述程序，其返回结果如图 6-11 所示。

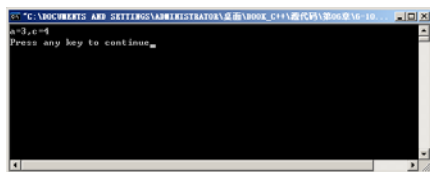


图 6-11 内联函数

【范例解析】读者可以看出，内联函数与普通的函数调用得到的结果是相同的，但是其内部的执行是不同的。调用内联函数时，编译系统直接将内联函数代码替换到主函数调用的地方，而普通函数则是通过参数传递，将函数的结果返回到主函数。



提示 由于编译时内联函数的代码将直接替换到主函数调用的地方，节省了调用传参数等步骤的时间，从而加快了运行速度。

6.2.7 宏与函数的区别

由于宏也可以带参数，而且带参数的宏与带参数的函数的写法和调用都很相似，但是其存在本质上的不同。前面已经提到过，函数调用时要把实参表达式的值求出来再赋予形参，而宏代换中对实参表达式不作计算直接地代换。这导致了即使把同一表达式用函数处理与用宏处理，两者的结果有可能是不同的。

【范例 6-11】宏的定义和调用与函数的定义和调用的比较。该范例定义了一个带参宏和带参函数，其函数名为 SQ，形参为 Y，函数体表达式为 $((y)*(y))$ ，而宏定义也定义字符串为 $((y)*(y))$ ，代码如代码清单 6-11 (a) 和代码清单 6-11 (b) 所示。

代码清单 6-11 (a)

```

1  //带参函数的定义和调用
2  #include <iostream.h>
3  int SQ(int);                      //声明函数 sq
4  void main()
5  {
6      int i=1;                      //定义变量并初始化
7      while(i<=5)                  //循环
8          cout<<SQ(i++)<<"\t";    //调用函数
9      cout<<endl;
10 }
11 int SQ(int y)                    //定义函数
12 {
13     return((y)*(y));              //返回一个整数的平方
14 }
```

【运行结果】上述程序实现的是一个简单的打印出 1~5 各个数字的平方值，在 Visual C++ 中运行该程序，其结果如图 6-12 所示。

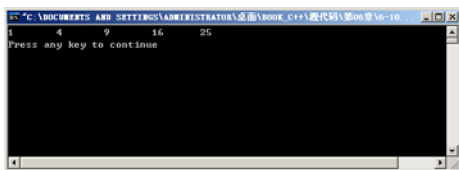


图 6-12 带参函数



【范例解析】该范例用函数的方法实现了输出 1~5 各个数字的平方值，其首先声明函数 SQ，再在主函数 main()中调用该函数。

代码清单 6-11 (b) 则实现带参数的宏的定义和调用，读者可将代码清单 6-11 (a) 中的函数的定义和调用与下面的宏的定义和调用相比较。

代码清单 6-11 (b)

```
1 //带参数
2 #define SQ(y) ((y)*(y)) //定义带参宏 sq
3 #include <iostream.h>
4 void main()
5 {
6     int i=1; //定义变量并初始化
7     while(i<=5) //循环
8         cout<<SQ(i++)<<"\t"; //调用该宏
9     cout<<endl;
10 }
```

【运行结果】该示例中使用的是带参数的宏定义，并在输出时调用该宏，同样，在 Visual C++ 中运行该程序，其结果如图 6-13 所示。

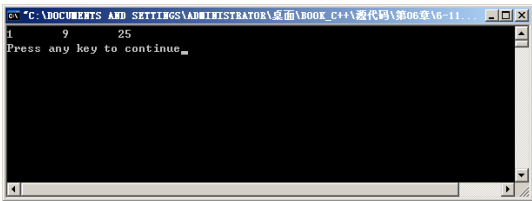


图 6-13 带参宏

【范例解析】读者可以看到，在上述两个示例中，代码 6-11 (a) 中定义的函数名为 SQ，形参为((y)*(y))；代码 6-11 (b) 中定义的宏名为 SQ，形参也为 y，字符串表达式为((y)*(y))，这两个代码是相同的。此外，代码 6-11 (b) 中调用函数为 SQ(i++)，代码 6-11 (b) 中的宏调用为 SQ(i++)，实参也是相同的。

然而从输出结果来看，使用函数和使用带参数的宏得到的结果却大不相同。这是因为在代码清单 6-11 (a) 中，函数调用是把实参 i 值传给形参 y 后自增 1，然后输出函数值，因而要循环 5 次。输出 1~5 的平方值。而在代码清单 6-11 (b) 中宏调用时，只作代换。SQ(i++)被代换为((i++)*(i++))。每次循环后 i 的值会增加 2，因此只做 3 次循环。

注意 读者从上述两段代码及其以上分析可以看出，函数调用和宏调用二者在形式上相似，但在本质上是完全不同的。

6.3 文件包含

除了 6.2 节介绍的宏定义外，文件包含是 C++ 预处理程序的另一个重要功能。文件包含是指一个 C++ 源程序中将另一个 C++ 源程序包含进来，通过 #include 预处理指令实现。

6.3.1 #include 命令

C++ 中，#include 被称为文件包含命令，其意义是把尖括号 < > 或引号 “ ” 内指定的文件包含到本程序来，成为本程序的一部分。被包含的文件通常是由系统提供的，其扩展名为 .h。

因此也称为头文件或首部文件。

C++的头文件中包括了各个标准库函数的函数原型，因此，凡是在程序中调用一个库函数时，都必须包含该函数原型所在的头文件。例如，要在 C++程序中使用流输入/输出命令 `cin` 和 `cout`，就必须包含 `iostream.h` 头文件，要使用 `printf` 和 `scanf` 输入/输出语句，则必须包含 `stdio.h` 头文件。头文件包含的库函数不要求读者都记住，只需熟悉几个常见库函数所在的头文件即可。

简单来说，文件包含命令的一般形式为：

```
#include "文件名"
```

或者

```
#include <文件名>
```



注意 `#include` 包含命令后的文件名可以用双引号括起来，也可以用尖括号括起来。例如以下写法都是允许的：

```
#include <iostream.h>
#include "iostream.h"
```

但是这两种形式是有区别的：使用尖括号表示在包含文件目录中去查找（包含目录是由用户在设置环境时设置的），而不在源文件目录去查找；使用双引号则表示首先在当前的源文件目录中查找，若未找到才到包含目录中去查找。了解了上述内容后，用户在编程时可根据自己文件所在的目录来选择某一种命令形式。

文件包含命令的功能是把指定的文件插入该命令行位置取代该命令行，从而把指定的文件和当前的源程序文件连成一个源文件。在程序设计中，文件包含是很有用的。一个大的程序可以分为多个模块，由多个程序员分别编程。有些公用的符号常量或宏定义等可单独组成一个文件，在其他文件的开头用包含命令包含该文件即可使用。这样，可避免在每个文件开头都去书写公用量，从而节省时间，并减少出错。

读者在使用 `#include` 文件包含命令时要注意如下两个事项：

- 一个 `include` 命令只能指定一个被包含文件，若有多个文件要包含，则需用多个 `include` 命令。
- 文件包含允许嵌套，即在一个被包含的文件中又可以包含另一个文件。

6.3.2 合理使用文件包含

在实际软件开发的时候，需要多个人构成的小组共同完成代码的编写与测试，因此需要借鉴和应用其他的结果，或者要借鉴前人的成果，这就需要用到文件包含了。

简单来说，文件包含的作用是在系统编译之前，将包含文件中的内容复制到当前文件的当前位置之后，再进行编译。

【范例 6-12】 文件包含的实现。该范例定义了两个 C++程序 `file1.cpp` 和 `file2.cpp`，在 `file2.cpp` 中需要用到 `file1.cpp` 中的函数，因此需要在 `file2.cpp` 中将 `file1.cpp` 包含进来，其实现代码如代码清单 6-12 所示。

代码清单 6-12

```
1 //file1.cpp
2 double Add(double a,double b)           //file1 文件中包含的函数
3 {
4     return a+b;                          //返回两个浮点数之和
5 }
6 //file2.cpp
7 #include "file1.cpp"                     //包含 file1.cpp 文件
```



```
8  #include <iostream.h>
9  void main()
10 {
11     double a,b;                //定义变量
12     double e,f;
13     a=3;                       //初始化变量
14     b=2;
15     e=Add(a,2);                //调用file1.cpp 文件中的函数Add
16     f=Add(b+1,2);              //调用函数
17     cout<<e<<"\t"<<f<<endl;  //输出结果
18 }
```

【运行结果】读者可以看出，上述代码的功能是求两个数之和，其中求和的函数放在 file1.cpp 文件中，而主函数 main()则在 file2.cpp 文件中，该程序段执行结果如图 6-14 所示。

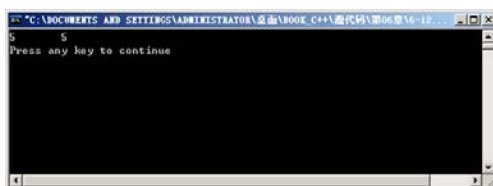


图 6-14 文件包含

【范例解析】上述示例中，在编译 file2.cpp 的时候，系统根据#include "file1.cpp"指令将 file1.cpp 的内容复制到当前文件的当前位置，所以在 file2.cpp 可以直接使用 double Add(double a,double b)函数。



提示 一般来说，在软件开发中将符号常量、全局变量、函数声明包含在头文件（.h 文件）中，并将其定义放在.cpp 文件中。然后在使用的时候，包含对应的头文件即可。

如果在程序中需要使用数学库函数，在文件中加入如下的代码即可：

```
#include <math.h>
```

或

```
#include "math.h"
```

#include <math.h>与#include "math.h"的区别在于，遇到#include <math.h>命令时系统从默认的头文件目录中查找文件 math.h 文件；而遇到#include "math.h"时系统首先从当前的目录中搜索，如果没有找到再在默认的头文件目录中查找文件 math.h 文件。因此包含系统提供的库函数使用#include <math.h>方式搜索速度比较快；如果包含用户自定义的.h 文件使用#include "math.h"方式，搜索速度比较快。



提示 在使用#include 指令的时候，对系统文件，使用#include <>的形式较好；而对用户自定义文件，则使用#include ""的形式速度较快。

6.4 条件编译

预处理程序除了提供上面介绍的宏定义和文件包含功能，其还提供了条件编译的功能。条件编译可以按不同的条件去编译不同的程序部分，因而产生不同的目标代码文件。这对于程序的移植和调试是很有用的。C++中的条件编译有三种形式，下面分别介绍。

6.4.1 #ifdef 形式

`#ifdef` 形式是指该形式的第一个编译命令为 `#ifdef`，这种形式的结构如下所示：

```
#ifdef 标识符
    程序段 1
#else
    程序段 2
#endif
```

该形式的条件编译功能是，如果标识符已被 `#define` 命令定义过，对程序段 1 进行编译，否则对程序段 2 进行编译。如果没有程序段 2（其为空），则本形式中的 `#else` 可以没有，即其形式可以改写为如下：

```
#ifdef 标识符
    程序段
#endif
```

【范例 6-13】 `#ifdef` 预编译命令的应用。该范例根据是否定义了 `PI` 来编译不同的程序块，如果定义了则编译计算圆面积的程序块，否则编译计算正方形面积的程序块，实现代码如下清单 6-13 所示。

代码清单 6-13

```
1  #define PI 3.1415926                                //定义宏 PI
2  #include <iostream.h>
3  void main()
4  {
5      double radius,sr,a,ss;                          //定义浮点型变量
6  #ifdef PI                                           //使用预编译命令#ifdef
7      {
8          cout<<"Please input radius:"<<endl;
9          cin>>radius;                                //接收键盘输入
10         sr=PI*radius*radius;                        //调用宏 PI
11         cout<<"The circle area is:"<<"\t"<<sr<<endl; //输出结果
12     }
13 #else
14     {
15         cout<<"Please input a:"<<endl;
16         cin>>a;                                       //接收键盘输入
17         ss=a*a;                                       //输出两个数的平方
18         cout<<"The square area is:"<<"\t"<<ss<<endl; //输出结果
19     }
20 #endif
21 }
```

【运行结果】 由于在程序的第 6 行插入了条件编译预处理命令，因此系统将根据 `PI` 是否被定义过来决定编译哪一个程序块的语句。而在程序的第 1 行已对 `PI` 做过宏定义，因此应对第一个程序块的语句做编译，因此运行结果如图 6-15 所示。

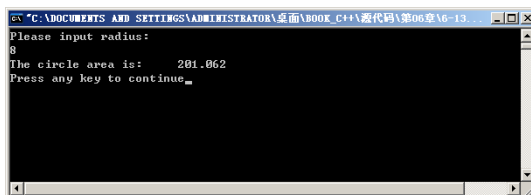


图 6-15 计算圆面积



在上述程序中，如果将第 1 行语句注释掉，那么系统将编译第二个程序块语句，即计算正方形的面积，而不是圆面积，运行结果如图 6-16 所示。

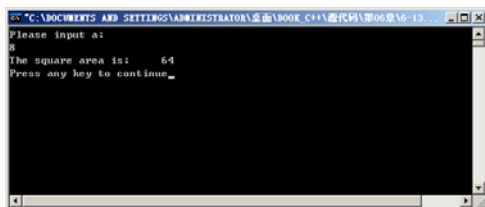


图 6-16 计算正方形面积

注意 在编译上述程序段时，不管是否将第 1 行代码注释，系统都会给出两个警告信息，这是正常现象。如果读者想消除这个警告，可以将定义变量的语句放在各个程序块中。

【范例解析】上述代码中，当定义了变量 PI 时，第二个程序段将不会被编译，因此变量 a 和 ss 没有被用到，系统将给出警告信息；而当注释第 1 行代码，也即没有定义 PI，则第一个程序段不会被编译，因此变量 radius 和 sr 将不会被用到，系统也将给出警告信息。一般来说，此处可以忽略该警告信息，直接运行程序即可。

6.4.2 #ifndef 形式

#ifndef 形式是指该形式的第一个编译命令为#ifndef，这种形式的结构如下所示：

```
#ifndef 标识符
    程序段 1
#else
    程序段 2
#endif
```

#ifndef 形式与第一种形式的区别是将“ifdef”改为“ifndef”。其功能是，如果标识符未被#define 命令定义过，则对程序段 1 进行编译，否则对程序段 2 进行编译。读者可以看出，#ifndef 形式与第一种形式的功能正相反。

提醒 关于这种形式的条件编译语句在具体程序中的应用，读者可以直接修改上述代码清单 6-14，看是否达到预期的目的，此处就不再赘述了。

6.4.3 #if 形式

#if 形式是指该形式的第一个编译命令为#if，结构如下所示：

```
#if 常量表达式
    程序段 1
#else
    程序段 2
#endif
```

这种形式的条件编译结构功能是，如常量表达式的值为真（非 0），则对程序段 1 进行编译，否则对程序段 2 进行编译。因此其可以使程序在不同条件下，完成不同的功能。

【范例 6-14】#if 预编译命令的应用。该范例将范例 6-13 进行修改，使其利用#if 形式达到同样的功能，其实现代码如代码清单 6-14 所示。

代码清单 6-14

```

1  #define flag 1 //定义宏 flag
2  #include <iostream.h>
3  void main()
4  {
5      #if flag //使用预编译命令#if
6      {
7          double radius,sr; //定义变量
8          cout<<"Please input radius:"<<endl;
9          cin>>radius; //接收键盘输入
10         sr=3.1415926*radius*radius; //计算圆面积
11         cout<<"The circle area is:"<<"\t"<<sr<<endl; //输出结果
12     }
13     #else //flag 值为 0 时
14     {
15         double a,ss;
16         cout<<"Please input a:"<<endl;
17         cin>>a; //接收键盘输入
18         ss=a*a; //计算正方形面积
19         cout<<"The square area is:"<<"\t"<<ss<<endl; //输出结果
20     }
21     #endif
22 }

```

【运行结果】将上述代码在 Visual C++ 中执行，其结果如图 6-17 所示。

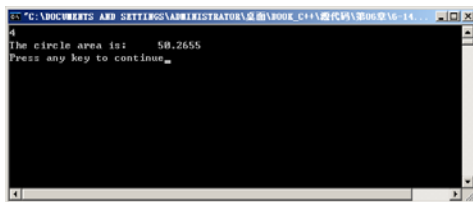


图 6-17 条件编译

【范例解析】读者看到，上述代码只是修改了第 1 行代码，定义一个无关的常量，在后续的代码中用 `#if` 来判断该常量是否非 0，是则执行程序块 1，否则执行程序块 2。上述代码中定义的常量 `flag` 为 1，执行计算圆面积的程序块。

当然，上面介绍的条件编译也可以用条件语句来实现。但是用条件语句将会对整个源程序进行编译，生成的目标代码程序很长，而采用条件编译，则根据条件只编译其中的程序段 1 或程序段 2，生成的目标程序较短。



提示 一般来说，在实际程序中，如果条件选择包含的程序段很长，采用条件编译的方法是十分必要的。

6.5 其他命令

前面介绍了预处理程序的许多命令，如宏定义、包含文件和条件编译等。除此之外，预处理程序还支持其他一些命令，这在本章一开始就列出了，本节将简要介绍其中的几个。

6.5.1 #error 命令

C++ 中，预处理程序中的 `#error` 指令用于程序的调试，在编译中遇到 `#error` 指令就停止编译。其一般形式如下：



`#error` 出错信息

需要注意的是，上述形式中的出错信息是不加引号的，当编译器遇到这个指令时，显示错误信息并停止编译。

C++提供`#error` 命令的目的是保证程序是按照用户所设想的那样进行编译的。系统编译程序时，只要遇到`#error` 就会跳出一个编译错误，用户就可以知道程序是否正常执行。例如，程序中往往有很多的预处理指令：

```
#ifdef XXX
...
#else
...
#endif
```

当程序比较大时，往往有些宏定义是在外部指定的，或是在系统头文件中指定的，当用户不太确定当前是否定义了某一个宏时，就可以改成如下这样进行编译：

```
#ifdef XXX
...
#error "XXX has been defined"
#else
...
#endif
```

这样，如果编译时出现错误，输出了 `XXX has been defined` 的错误提示，那就表明宏 `XXX` 已经被定义了。

6.5.2 `#line` 命令

`#line` 命令用于控制行号，其一般在发布错误和警告信息时使用。当用户在编译一段程序的时候，如果有错误发生，编译器会在错误前面显示出错文件的名称及文件中的第几行发生错误。指令`#line` 可以实现这个功能，也就是说，当出错时显示文件中的行数及希望显示的文件名。该命令的格式是：

```
#line number "filename"
```

此处的 `number` 是将会赋给下一行的新行数，其后面的行数从这一点逐个递增。`filename` 是一个可选参数，用来替换自此行以后出错时显示的文件名，直到有另外一个`#line` 指令替换它或直到文件的末尾。例如：

```
#line 1 "assigning variable"
int a?;
```

这段代码将会产生一个错误，显示为在文件"assigning variable", line 1 。

6.6 小结

本章主要介绍了 C++中编译预处理的基本内容。结合 C++中使用较多的地方，本章对宏及其相关应用做了详细讲解，依次介绍了宏的定义、调用、无参宏和带参宏的定义调用，以及宏与函数的区别等，对于难点部分，都安排了具体示例方便读者理解。此外，本章还对包含文件处理`#include` 命令、条件编译相关命令等做了简要的介绍。学习完本章，读者应对编译器编译 C++源程序的过程有一定理解，并了解如何优化程序的部分方法。

6.7 习题

1. 什么是预编译，何时需要预编译？

【解答】预编译就是指程序执行前的一些预处理工作，主要指用#来表示的一系列表达式。使用预编译主要针对：

- 总是使用不经常改动的大型代码体。
- 程序由多个模块组成，所有模块都使用一组标准的包含文件和相同的编译选项。在这种情况下，可以将所有包含文件预编译为一个预编译头。

2. 编写一个标准宏 MIN，这个宏输入两个参数并返回较小的一个，注意参数在调用时可能是表达式的情况。

【解答】该习题主要考查宏定义的定义和使用。宏定义可以实现类似于函数的功能，但是它终究不是函数，而宏定义中括号中的“参数”也不是真的参数，在宏展开的时候对“参数”进行的是一对一的替换。读者需要谨慎地将宏定义中的“参数”和整个宏用括号括起来，所以严格地讲下述解答都是错误的：

```
#define MIN(A,B) (A) <= (B) ? (A) : (B)
#define MIN(A,B) (A <= B ? A : B )
```

正确的写法应如下所示：

```
#define MIN(A,B) ((A) <= (B) ? (A) : (B))
```

3. 编写一个 C++ 程序，在程序中定义一个不带参数的宏 PI，使其完成求给定半径的圆的周长和面积。

【解答】该习题主要考查宏在具体程序中的应用。根据前面章节的学习，读者知道求给定圆半径的周长和面积可以通过声明变量和常量来实现，其中由于 PI 是不会变的，其值为 3.14，因此可以将其声明为常量。在学习了宏后，可以将该常量声明为宏，其他语句不变。其简要实现代码如下所示。

```
#include <iostream.h>
#define PI 3.14
void main()
{
    float radius;
    double area,len;
    cout<<"请输入圆半径: ";
    cin>>radius;
    len=2*PI*radius;
    area=PI*radius*radius;
    cout<<"圆周长为: "<<len<<endl;
    cout<<"圆面积为: "<<area<<endl;
}
```

4. 下面程序段定义了两个宏，在主函数 main() 中使用了条件编译语句来控制程序的运行，读者仔细理解并写出输出结果。

```
#include<iostream.h>
#define CIR(r) r*r //带参数的宏定义
#define TEST //定义宏
void main()
{
    int x=1; //定义并初始化变量
    int y=2;
    int z;
    z=CIR(x+y); //调用宏
    cout<<"CIR(x+y)= "<<z<<endl; //输出宏调用的结果
    #ifdef TEST //条件编译语句
    cout<<"x= "<<x<<"\t"<<"y= "<<y<<endl; //输出结果
    #endif //结束条件编译
```



```
    cout<<"z= " <<z<<endl;           //输出结果
}
```

【解答】上述代码中，定义了带参数的 CIR(r)，其主函数 main() 中展开后为 $z=x+y*x+y$ ，将 x 和 y 的值代入，因此 $z=5$ 。而条件编译语句 `#ifdef TEST` 表示如果 TEST 已被定义则执行下边的语句，因此程序将输出 x、y 和 z 的值，输出结果如图 6-18 所示。

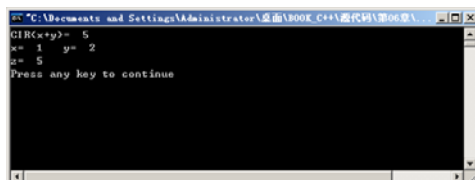


图 6-18 编译预处理

5. 宏可以带参数，而且带参数的宏与带参数的函数的写法和调用都很相似，但是其存在本质上的不同，其区别在何处，请试着通过两个例子进行说明。

【解答】该习题主要考查函数和宏的区别。函数调用时要把实参表达式的值求出来再赋予形参，而宏代换中对实参表达式不作计算，直接地代换。这导致了即使把同一表达式用函数处理与用宏处理，两者的结果也有可能是不同的。下面通过具体例子进行说明。如下分别定义了一个带参宏和带参函数，其函数名为 SQ，形参为 Y，函数体表达式为 $((y)*(y))$ ，而宏定义也定义字符串为 $((y)*(y))$ 。

```
//带参函数的定义和调用
#include <iostream.h>
int SQ(int);           //声明函数 SQ
int main()
{
    int i=1;           //定义变量并初始化
    while(i<=5)        //循环
        cout<<SQ(i++)<<"\t"; //调用函数
    cout<<endl;
}
int SQ(int y)          //定义函数
{
    return((y)*(y));   //返回一个整数的平方
}

//带参宏
#define SQ(y) ((y)*(y)) //定义带参宏 SQ
#include <iostream.h>
void main()
{
    int i=1;           //定义变量并初始化
    while(i<=5)        //循环
        cout<<SQ(i++)<<"\t"; //调用该宏
    cout<<endl;
}
```

从输出结果来看，使用函数和使用带参数的宏得到的结果却大不相同。这是因为在函数定义代码中，函数调用是把实参 i 值传给形参 y 后自增 1，然后输出函数值，因而要循环 5 次。输出 1~5 的平方值。而在宏定义代码中宏调用时，只做代换。SQ(i++)被代换为 $((i++)*(i++))$ 。每次循环后 i 的值会增加 2，因此只做 3 次循环。

第 7 章 数组

在任何一种程序设计语言中，数组都是非常重要的一种类型，在 C++ 语言中同样如此。与普通的数据类型不同，数组类型是一种构造型（复合型）的数据类型。

一般情况下，一个数组中的元素类型必须相同，可以是前面讲过的各种基本类型。一个数组可以是一维的，也可以是多维的。一般二维数组用于表示一个平面内需要两个坐标来表示的元素；而三维数组用于表示一个立体空间内需要三个坐标来表示的元素。以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为一个学习的参照。

- 了解数组的概念。
- 熟练掌握一维和多维数组的声明与引用。
- 掌握数组的多种赋值方法。
- 熟悉数组在实际程序中的应用。

7.1 声明数组

简单地说，数组就是由一些具有相同数据类型元素组成的集合，这些元素在内存中占用一组连续的存储单元，而数组的类型就是这些元素的数据类型。在程序设计语言中，用一个统一的名称标识这一组数据，即数组名。

严格来说，数组并不是一种数据类型，而是一组相同类型的变量的集合。在程序中使用数组的好处是可以用一个统一的数组名代表逻辑上相关的一组数据，并用下标表示各元素在数组中的位置。比如，在汽车生产车间，依次摆着 10 辆新生产的汽车，其编号分别为第 1 辆、第 2 辆……那么这就是一个汽车数组，可以表示为汽车 [1]、汽车 [2] ……如图 1-1 所示。

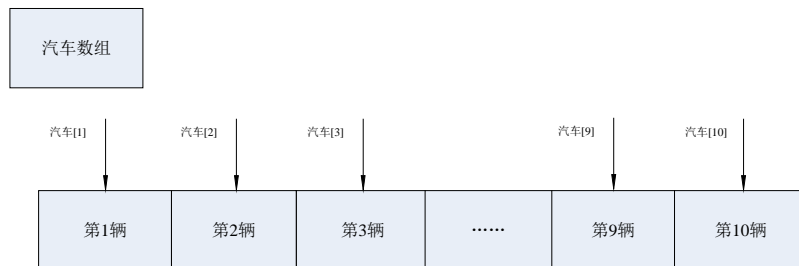


图 7-1 数组的概念

但是需要注意的是，C++ 中数组元素的编号是从 0 开始的，这在本节的后面将详细讲解。

7.1.1 声明一维数组

和普通的变量一样，数组在使用前都必须先声明。数组的声明分为一维数组和多维数组的声明，这是根据数组的分类来区分的。一维数组在具体程序中使用是非常广泛的，在使用一维数组前，先简单看一下其声明。一维数组的声明一般形式为：

<类型名><数组名>[<下标表达式>]={<初值表>};



其中，各部分的含义如下。

- 类型名：类型名表示数组类型，即数组中各元素的数据类型，可以是整型、浮点型、字符型等基本类型，也可以是基本类型的派生类型名、类名、枚举类型名、结构、联合类型名。
- 数组名：数组名是一个标识符，表示数组元素在内存中的起始位置，通常是个常量，不能赋值，代表着数组元素在内存中的起始地址，其命名规则与变量名的命名一样。
- 下标表达式：下标表达式是一种表达式，很多情况下其是一个正整数，表示数组的大小，即一维数组中元素的个数。下标表达式要用方括号“[]”括起来。方括号“[]”的个数代表数组的维数，一个方括号表示一维数组。方括号不可默认，下标表达式可缺省，但必须对数组赋初值，系统会根据所赋的初值个数来确定数组的大小。
- 初值表：是可缺省的，并不是声明数组时就需要赋初值。如果在声明时赋初值，那么初值需用花括号“{ }”括起来，并用逗号“,”分隔每个初值。

例如，下面分别定义了一个没有赋初值的具有 5 个元素的一维字符型数组 a 和一个具有 10 个元素的一维整型数组 b。

```
char a[5];
int b[10];
```

对上面定义的数组 b，也可以采用下面这种定义方法：

```
const int size=10;
int b[size];
```

上述方法中，数组的下标并不是一个整数，而是一个表达式。



提示 下标为表达式时，该表达式只能为一个常量。也即在定义数组时，不能用变量来描述数组定义中的元素个数。例如，下面的定义方式是不合法的：

```
int b[n];
```

数组元素在内存中是顺序存储的。对于一维数组，就是简单地按下标顺序存储。例如，对上面定义的整型数组 b，在内存中的存放顺序如图 7-2 所示。

7.1.2 声明多维数组

多维数组是在一维数组声明方式的基础上，增加下标的维数，即增加[]的个数，声明格式中定义了 n 个[]，就表示 n 维数组。一般来说，多维数组的声明格式如下所示：

<类型名><数组名>[<下标表达式 1>][<下标表达式 2>]...[<下标表达式 n>]

多维数组在实际中应用最广泛的是二维数组，即声明数组时其“[]”的个数为 2。二维数组在数学中也称为矩阵，需要两个下标才能标识某个元素的位置，通常称第一个下标为行下标，称第二个下标为列下标。声明二维数组的语法格式可以简化为：

类型 数组名[常量表达式 1][常量表达式 2];

定义二维数组的格式与定义一维数组的格式相同，只是必须指定两个常量表达式，第一个常量表达式标识数组的行数，第二个常量表达式标识数组的列数。例如：

```
int a[2][3];
```

上面定义了一个数组 a，它在逻辑上的空间形式为 2 行 3 列，每一个数组元素都是整型数

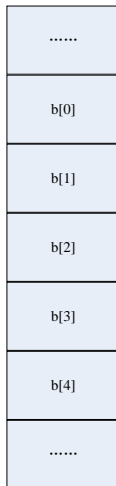


图 7-2 一维数组的存储

据, 因此 a 数组的各元素如下:

```
a[0][0]  a[0][1]  a[0][2]
a[1][0]  a[1][1]  a[1][2]
```

二维数组在内存中的排列顺序是“先行后列”, 即在内存中先存第一行的元素, 然后再存第二行的元素。从数组下标变化来看, 先变第二个下标, 第一个下标先不变化(即 `i[0][0]`, `i[0][1]`, `i[0][2]`), 待第二个下标变到最大值时, 才改变第一个下标, 第二个下标又从 0 开始变化。例如, 下列几条语句分别对不同类型的数组进行了声明。

```
char a[10];           //声明一个长度为 10 的一维字符型数组 a
int b[]={1,2,3};      //声明一维数组 b, 并给数组的第 0~2 个元素依次赋值为 1, 2, 3
float c[2][3];        //声明一个二维单精度浮点型数组 c, 该数组有 2 行 3 列
```

7.2 引用数组

在声明数组时用数组名表示该数组的整体, 但 C++ 语言没有提供对数组进行整体操作的运算符和运算, 而针对每个数组元素进行操作时, 数组元素一般是通过下标变量来区分的, 这就涉及在具体应用中如何引用数组元素的问题。

7.2.1 引用一维数组

同样, 根据数组的分类, 对数组的引用也可分为一维数组的引用和二维数组的引用。本节将介绍一维数组的引用。一般来说, 一维数组的数组元素引用的一般形式为:

<数组名>[<下标>]

其中, 下标指明了数组中每个元素的序号, 值为整数, 用数组名加下标值就可以访问数组中对应的某个元素。



注意 下标值是从 0 开始的, 因此对于一个具有 n 个元素的一维数组来说, 它的下标值是 0 ~ n-1。

例如, 对用 `int b[10]` 定义的数组 b 来说, `b[0]` 是数组中的第一个元素, `b[1]` 是数组中的第二个元素, 依此类推, `b[9]` 是数组中的最后一个元素。读者可以看出, 一个声明为 `int b[10]` 的一维数组中是不包含元素 `b[10]` 的。一维数组在实际程序中的应用非常广泛。

【范例 7-1】 引用一维数组。该范例实现从键盘接收 5 个数字, 并将其按照输入顺序倒序输出, 实现代码如代码清单 7-1 所示。

代码清单 7-1

```
1  #include <iostream.h>           //包含输入/输出头文件
2  void main()
3  {
4      int a[5];                   //声明包含 5 个元素的一维数组
5      int i, j;                   //定义两个整型变量
6      cout<<"Please input 5 number:"<<endl;
7      for(i=0; i<5; i++)          //循环接收输入
8          cin>>a[i];
9      cout<<"convert into:"<<endl;
10     for(j=4; j>=0; j--)          //循环倒序输出
11         cout<<a[j]<<"\t";        //中间隔开一个 Tab 空格
12     cout<<endl;                 //输出换行
13 }
```



【运行结果】在 Visual C++ 中新建一个【C++ Source File】，将上述代码输入其中，编译无误后运行，其结果如图 7-3 所示。

【范例解析】上述程序段声明了一个包含 5 个元素的整型一维数组 `a[5]`，首先通过 `for` 循环接收用户输入 5 个数值，将其存入数组 `a` 中；接下来用 `for` 循环将其中数据输出。读者可以看出，输入 5 个数组元素时，循环变量 `i` 的取值为 0~4，注意不能出现数组元素 `a[5]`。同样，在输出时，循环变量 `j` 的取值为 4~0。

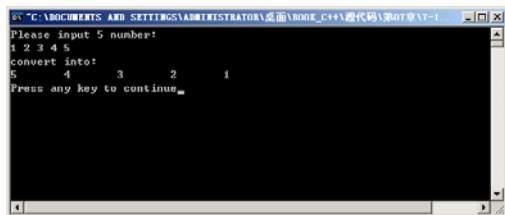


图 7-3 引用一维数组

7.2.2 引用多维数组

多维数组的引用与一维数组类似，其引用也根据下标的变化从 0~ $n-1$ 取值即可。与一维数组不同的是，多维数组含有多个[]，因此其中每个[]中的下标都需要从 0~ $n-1$ 变化。一般来说，一个 n 维数组的数组元素引用的一般形式为：

<数组名>[<下标 1>][<下标 2>]...[<下标 n >]

以二维数组为例，二维数组的数组元素引用形式为：

<数组名>[<行下标>][<列下标>]

二维数组是 multidimensional array 中应用最广泛的一种，二维数组的数组元素个数是其行和列的下标乘积。例如，二维数组 `a[3][4]`，其包含的数组元素个数为 $3 \times 4 = 12$ 个。



提示 在多维数组中，其存放方式为按行存储。例如，通过语句 `int b[3][4]` 声明一个整型二维数组 `b`，其数组元素分别如下：

```
b[0][0], b[0][1], b[0][2], b[0][3],
b[1][0], b[1][1], b[1][2], b[1][3],
b[2][0], b[2][1], b[2][2], b[2][3].
```

前面已经介绍过了，多维数组在内存中的存储次序为按行存储，因此，上述二维数组元素的排列即为其在内存中的存储次序。

【范例 7-2】引用多维数组。该范例从键盘接收 9 个数字，将其存入二维数组中，并以矩阵的形式输出，其实现代码如代码清单 7-2 所示。

代码清单 7-2

```
1  #include <iostream.h>
2  void main()
3  {
4      int a[3][3];                //声明二维数组
5      int i,j;
6      cout<<"Please input 9 number:"<<endl;
7      for(i=0;i<3;i++)            //进入双重循环
8          for(j=0;j<3;j++)
9          cin>>a[i][j];            //接收数组元素
```

```

10     cout<<"Output:"<<endl;
11     for(i=0;i<3;i++)                                //进入双重循环
12     {
13         for(j=0;j<3;j++)
14             cout<<a[i][j]<<"\t";                    //输出
15     cout<<endl;                                       //输出换行
16     }
17     cout<<endl;                                       //输出换行
18 }

```

【运行结果】上述代码在 Visual C++ 中，其执行结果如图 7-4 所示。

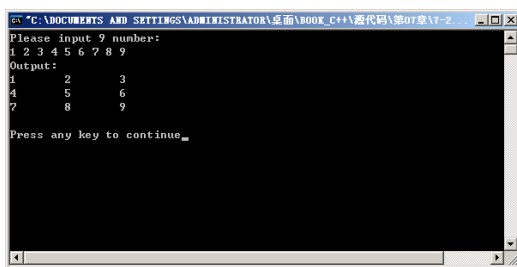


图 7-4 引用二维数组

【范例解析】上述代码中，定义了一个包含 9 个元素的二维数组 a，使用了一个双重循环接收用户的键盘输入，并将其存储在 a 的数组元素中，再用一个双重循环将其以矩阵的形式输出。



警告 为了提高程序的执行速度，C++ 语言对于数组的越界使用不作检查，所以编程时要注意引用数组元素时不要超过所定义的界限，否则将出现不可预知的错误，这就是数组下标越界问题。

上述代码中，如果输出时将 i 的取值设置小于 4，即允许其输出 a[2][3]，a[3][2]，a[3][3] 等几个值，其输出结果如图 7-5 所示。

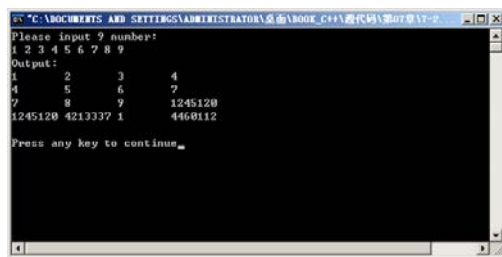


图 7-5 数组下标越界

读者可以看出，输出的数组元素中，凡是越界的都显示错误数值，而这个数值是不可预知的。因此，在引用数组元素时，一定要注意数组的下标不能越界。

7.3 数组的赋值

数组的赋值是对数组操作的一个重要部分，其主要包括初始化数组和在实际应用中对数组进行赋值。对数组的赋值方法较多，本节将重点介绍三种方法。

7.3.1 初始化数组

在定义数组时对其中的全部或部分指定初始值，这称为数组的初始化。只有存储类别为静



态的或外部的数组才可以进行初始化。初始化的语法格式为：

类型 数组名[数组范围]={值 1,值 2,⋯,值 n}

例如，下面代码定义了一个字符型数组 a，其中包含 5 个元素，在定义该数组的同时为其初始化，代码如下：

```
char a[5]={'a','b','c','d','e'};
```

或

```
char a[ ]={'a','b','c','d','e'};
```



提示 上述的初始化是指定了数组中所有元素的值。事实上，在对数组初始化时，也可以只对数组中的部分元素指定初始值。初始化值的个数可以少于或等于数组定义的元素的个数，但不可以多于数组元素的个数，否则会引起编译错误。

当初始化值的个数少于数组元素个数时，前面的元素按顺序初始化相应的值，后面不足的部分由系统自动初始化为零（对数值数组）或空字符'\0'（对字符数组）。例如，下面代码对一个包含 5 个元素的整型数组 c 初始化：

```
int c[5]={1,2};
```

上述代码定义整型数组 c 有 5 个元素，但只初始化前两个元素：c[0]=1，c[1]=2。对于后面的 3 个元素没有定义初始值，此时由系统自动给它们赋值为 0。



注意 当数组长度与初始化元素的个数不相等时，数组长度不能省去不写。如上例不能写为：

```
int c[ ]={1,2};
```

这种写法是错误的，这会让编译器认为数组 c 只有 2 个元素而不是 5 个元素。

【范例 7-3】 初始化数组。该范例定义了 4 个数组，并分别对其进行了初始化，根据初始化程度的不同，其数组的元素也不同，实现代码如代码清单 7-3 所示。

代码清单 7-3

```
1  #include <iostream.h>
2  void main()
3  {
4      char a[5]={'a','b','c','d','e'};    //定义包含 5 个元素的数组并全部初始化
5      char b[]={'a','b','c'};            //定义未知长度的数组并初始化
6      int c[5]={1,2};                    //定义包含 5 个元素的数组并部分初始化
7      char d[]={'a','b'};                //定义未知长度的数组并初始化
8      cout<<"a:";
9      for (int i=0;i<5;i++)              //输出数组 a
10         cout<<a[i]<<"\t";
11     cout<<endl;                        //输出换行
12     cout<<"b:";
13     for (i=0;i<5;i++)                  //输出数组 b
14         cout<<b[i]<<"\t";
15     cout<<endl;                        //输出换行
16     cout<<"c:";
17     for (i=0;i<5;i++)                  //输出数组 c
18         cout<<c[i]<<"\t";
19     cout<<endl;                        //输出换行
20     cout<<"d:";
21     for (i=0;i<5;i++)                  //输出数组 d
22         cout<<d[i]<<"\t";
```

```

23         cout<<endl;           //输出换行
24     }

```

在 Visual C++ 中执行上述代码, 其结果如图 7-6 所示。

【范例解析】从上述示例读者可以看出, 数组 a 有 5 个元素, 依次全部显示其初始化后的值; 数组 b 只有 3 个元素, 如果强制显示其 5 个元素, 则为乱码; 数组 c 有 5 个元素, 但只初始化前 2 个元素, 因此后两个元素自动赋值 0; 数组 d 有 2 个元素。

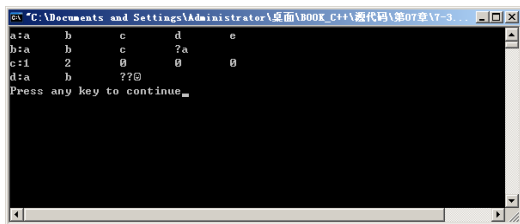


图 7-6 初始化数组

提示

从上述示例再次验证了在给数组元素赋值或对数组元素进行引用时, 一定要注意, 下标的值不要超过数组的范围, 否则会产生数组越界问题。因为当数组下标越界时, 虽然编译器并不认为它是一个错误, 但这往往会带来非常严重的后果。

7.3.2 通过赋值表达式赋值

前面章节提到的赋值表达式也可以给数组赋值, 这种方式既可以在初始化时给数组赋值, 也可以在程序运行过程中赋值。

注意

需要注意的是, 通过赋值表达式为数组赋值, 是直接给某个具体的数组元素赋值, 而不是给整个数组赋值。例如:

```

a[3]=4;

```

上述语句实现给元素 a[3] 赋值 4。而如下的写法则是错误的:

```

a=4;

```

【范例 7-4】通过赋值表达式赋值。该范例实现了通过赋值表达式分别对数组的各个元素赋值, 其代码如代码清单 7-4 所示。

代码清单 7-4

```

1  #include <iostream.h>           //包含输入/输出头文件
2  void main()
3  {
4      int a[3];                   //定义数组
5      a[0]=2;                     //通过赋值表达式给数组元素赋值
6      a[1]=4;
7      a[2]=6;
8      cout<<a[0]<<"\t"<<a[1]<<"\t"<<a[2]<<endl; //输出数组元素的值
9  }

```

【运行结果】该代码在 Visual C++ 的执行结果如图 7-7 所示。

【范例解析】上述程序中, 依次对定义的数组 a 中的每个元素进行赋值, 并在最后输出。读者可以看出, 通过赋值表达式对数组进行赋值的效率是非常低的, 因为其需要对数组中的每个元素进行赋值。

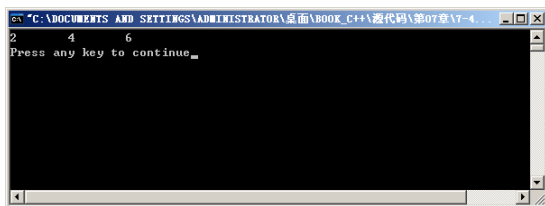


图 7-7 通过赋值表达式赋值

7.3.3 通过输入语句赋值

除了 7.3.2 节介绍的通过赋值表达式赋值外, C++中还可以通过输入语句对数组元素进行赋值, 这增加了数组赋值灵活性。

【范例 7-5】通过输入语句赋值。该范例从键盘接收用户输入的 3 个数值, 将其分别放在数组中, 其实现代码如代码清单 7-5 所示。

代码清单 7-5

```
1  #include <iostream.h>                                //包含输入/输出头文件
2  void main()
3  {
4      int a[3];                                          //定义包含 3 个元素的数组
5      cout<<"Please input 1st number: ";
6      cin>>a[0];                                         //给第一个数组元素赋值
7      cout<<"Please input 2nd number: ";
8      cin>>a[1];                                         //给第二个数组元素赋值
9      cout<<"Please input 3rd number: ";
10     cin>>a[2];                                         //给第三个数组元素赋值
11     cout<<a[0]<<"\t"<<a[1]<<"\t"<<a[2]<<endl;        //输出所有数组中的元素
12 }
```

【运行结果】在 Visual C++中执行上述程序的结果如图 7-8 所示。

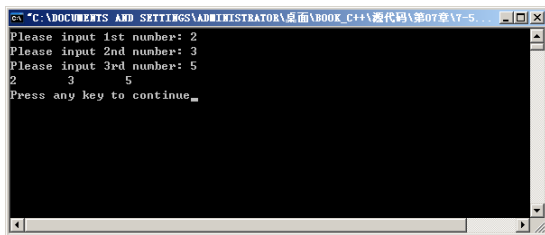


图 7-8 通过输入语句赋值

【范例解析】上述代码中, 依次接收用户输入的 3 个数值, 并将其按数组顺序依次存储到第一个位置、第二个位置和第三个位置, 最后输出该数组中的元素。



读者可以看出, 无论是通过赋值语句赋值和输入语句赋值, 都必须逐个给数组中的每个元素赋值, 当数组元素较多时, 这样做的效率是非常低的。

7.3.4 通过循环语句赋值

通过循环语句赋值可以缓解部分数组赋值效率低下的问题。需要注意的是, 通过循环语句赋值要求数值有一定的规律性。

【范例 7-6】通过循环语句赋值。该范例实现给一个数组赋值，其赋值的规律为：将 0 赋给数组的第一个元素，将 1 赋给数组的第二个元素，……将 9 赋给数组的最后一个元素，实现代码如代码清单 7-6 所示。

代码清单 7-6

```

1  #include<iostream.h>
2  #include<iomanip.h>           //包含头文件 iomanip.h
3  void main()
4  {
5      int a[10],i;
6      for(i=0;i<=9;i++)         //通过循环语句给数组赋值
7          a[i]=i;
8      for(i=9;i>=0;i--)         //循环倒序输出数组中元素
9          cout<<setw(4)<<a[i];   //使用 setw() 函数进行输出格式控制
10         cout<<endl;           //输出换行
11 }

```

【运行结果】在 Visual C++ 中运行上述代码，其执行结果如图 7-9 所示。



图 7-9 通过循环语句赋值

【范例解析】上述语句中，第一个 for 循环语句实现赋值，第二个 for 循环实现将刚赋值的结果输出，以便读者验证。

【范例 7-7】循环语句与输入语句相结合赋值。该范例通过循环语句和输入语句的结合，实现按照用户的输入依次给数组中的各元素赋值，代码如代码清单 7-7 所示。

代码清单 7-7

```

1  #include <iostream.h>         //包含输入/输出头文件
2  void main()
3  {
4      int a[4];                 //定义整型数组
5      int i;
6      for(i=1;i<4;i++)          //for 循环语句
7          cin>>a[i];            //与输入语句相结合
8      cout<<"Output: ";
9      for(i=1;i<4;i++)          //循环输出
10         cout<<a[i]<<"\t";      //输出数组中所有元素的值
11     cout<<endl;
12 }

```

【运行结果】同样，在 Visual C++ 中运行上述代码，其执行结果如图 7-10 所示。

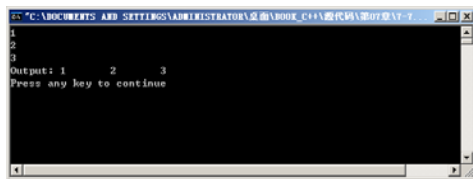


图 7-10 循环输入赋值



【范例解析】上述代码接收用户循环输入的3个数值，并将其依次赋值给 a[1]，a[2]，a[3]，最后将其输出。



上述程序段中，数组元素 a[0] 没有赋值，即 a[0] 中存储的是一个不可知的值。因此，在使用中，不能调用 a[0] 元素。

7.3.5 多维数组的赋值

前面讲解的内容都是针对一维数组而言的，在本节中将介绍多维数组，尤其是二维数组的赋值。二维数组的赋值与一维数组类似，但采用循环输入为其赋值时需要使用双重循环。

【范例 7-8】多维数组的赋值。该范例实现从键盘接收用户输入给数组赋值，并将赋值后的数组显示出来，代码如代码清单 7-8 所示。

代码清单 7-8

```
1  #include <iostream.h>           //包含输入/输出头文件
2  void main()
3  {
4      int a[2][3];                 //定义含有 6 个元素的二维数组 a
5      int i,j;                     //定义整型变量
6      cout<<"请输入 6 个数组元素:"<<endl;
7      for (i=0;i<2;i++)           //行、列循环接收输入
8          for (j=0;j<3;j++)
9              cin>>a[i][j];        //数组赋值
10     cout<<endl<<"输入的数组如下:"<<endl;
11     for (i=0;i<2;i++)           //输出数组
12     {
13         for (j=0;j<3;j++)
14         {
15             cout<<a[i][j];        //输出二维数组元素
16             cout.width(5);        //控制输出格式，宽度为 5
17         }
18         cout<<endl;              //换行
19     }
20 }
```

【运行结果】在 Visual C++ 中运行上述代码，其执行结果如图 7-11 所示。

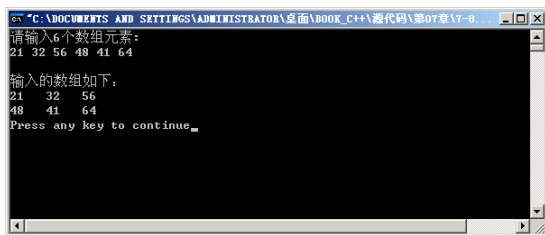


图 7-11 二维数组的赋值

【范例解析】上述代码中，由于定义的是一个二维数组，该数组中含有 6 个元素。使用了两个双重循环，第一次是用于接收用户输入，第二次是输出数组。

7.4 字符串

在现实世界中，许多值都需要使用一串字符来表示，这就是程序设计语言中所说的字符串。

在许多程序设计语言中,字符串作为一种基本的数据类型。在 C++ 中,字符串可以通过两种方式表示:传统字符串和字符数组。

7.4.1 传统字符串

字符串是 C++ 中应用广泛的一个数据类型。例如,下面语句定义了两个字符串:

```
char ch1[]={"Welcome"}
char ch2[]={"to China"}
```

由于字符串的重要性,许多编程语言都提供了关于字符串处理的一些函数,C++ 也是如此。C++ 的字符串标准函数的原型在头文件 string.h 中,其常用的几个函数如下。

1. strcpy()函数

strcpy()函数用于复制字符串,其主要功能是将源字符串复制到目标字符串,调用的形式如下:

```
strcpy(char destination[], const char source[]);
```

【范例 7-9】strcpy()函数的应用。该范例调用 strcpy()函数对上述定义的 2 个字符串 ch1 和 ch2 进行字符串的复制操作,其实现代码如代码清单 7-9 所示。

代码清单 7-9

```
1  #include <iostream.h>                                //包含输入/输出头文件
2  #include <string.h>                                    //包含头文件
3  void main()
4  {
5      char ch1[]={"Welcome"};                          //定义字符串
6      char ch2[]={"to China"};
7      cout<<"The result is: "<<endl<<strcpy(ch1,ch2)<<endl;
8                                     //调用 strcpy()函数复制字符串
9  }
```

【运行结果】上述代码中,将对字符串 ch1 和字符串 ch2 进行 strcpy 操作后的结果输出,如图 7-12 所示。

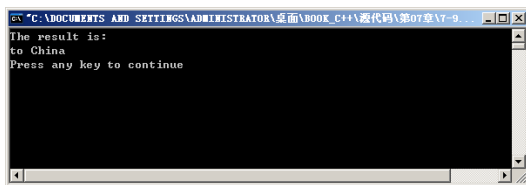


图 7-12 strcpy()函数

【范例解析】从该示例读者可以看出, strcpy()函数源字符串 ch2 中的内容“to China”复制到目标字符串 ch1 中,并输出复制结果。

注 目标字符串必须定义得足够大,以便容纳复制过来的字符串。上述示例中, ch2 恰好是大于 ch1 的,否则需要用户自定义 ch1 和 ch2 的长度,保证 ch2 的长度大于 ch1。

2. strcat()函数

strcat()函数用于连接字符串,主要功能是将字符串 source 连接到字符串 target 的后面,调用的形式为:

```
strcat(char target[], const char source[]);
```



【范例 7-10】strcat()函数的应用。该范例调用 strcat()函数对上述定义的 2 个字符串 ch1 和 ch2 进行字符串连接操作，其实现代码如代码清单 7-10 所示。

代码清单 7-10

```
1  #include <iostream.h>                                //包含输入/输出头文件
2  #include <string.h>                                    //包含字符串处理头文件
3  void main()
4  {
5      char ch1[]={" to China"};                          //定义字符串并初始化
6      char ch2[]={"Welcome"};
7      cout<<strcat(ch2,ch1)<<endl;                      //调用 strcat()函数连接字符串
8  }
```

【运行结果】在 Visual C++中执行上述代码，其结果如图 7-13 所示。

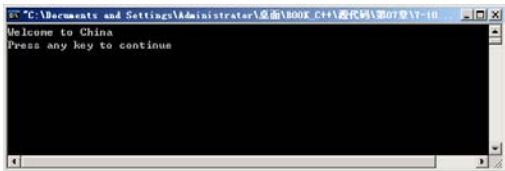


图 7-13 strcat()函数

【范例解析】上述代码中，将对字符串 ch1 和字符串 ch2 进行 strcat 操作后的结果输出，即将 ch1 中的字符串连接到 ch2 字符串后。

3. strlen 函数

strlen()函数用于计算字符串的实际长度，其调用的一般形式为：

```
strlen( const char string[] );
```

【范例 7-11】strlen 函数的使用。该范例计算出字符串 ch 的长度，并将其输出，其实现代码如代码清单 7-11 所示。

代码清单 7-11

```
1  #include <iostream.h>
2  #include <string.h>                                //包含头文件 string.h
3  void main()
4  {
5      char ch[]={"Welcome"};                          //定义字符串
6      int num;                                          //定义整型变量
7      num=strlen(ch);                                  //调用 strlen 求字符串长度
8      cout<<"The length of ch is : "<<num<<endl;      //输出结果
9  }
```

【运行结果】上述程序在 Visual C++中的运行结果如图 7-14 所示。

【范例解析】上述代码中，定义了一个字符串 ch，并为其初始化，接着定义了一个变量 num 用于存储求出的字符串 ch 的长度，最后输出。

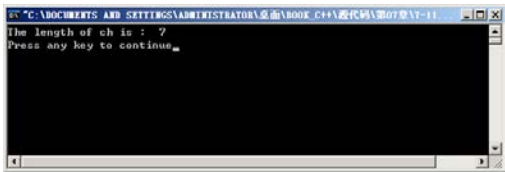


图 7-14 strlen 函数



注意 凡是使用了字符串处理函数,在预编译部分就必须包含头文件,即加入包含语句#include <string.h>,否则编译时将出现错误。

除了上述介绍的三种常用函数外,C++还提供了诸如 strlwr()函数(将字符串中大写字母转换成小写)、strupr()函数(将字符串中小写字母转换成大写)、strcmp()函数(字符串大小比较)等函数,此处不再一一讲解,有兴趣的读者可参阅相关资料。

7.4.2 字符数组

从表面上看,一个字符串是一个字符数组,但在 C++ 语言中,它们是不完全相同的。字符串是一个以空字符“\0”作为结束符的字符型数组。例如,下列定义字符数组的语句所代表的含义是不同的:

```
char a[]={"Hello"};           //声明一个字符串,其长度为6,包括5个字符和一个'\0'结束符。
char b[]={'H','e','l','l','o'}; //声明一个字符数组,其长度为5,包括5个字符。
```

【范例 7-12】字符数组的应用。该范例从键盘上接收用户不超过 10 个字符的输入字符,以输入字符 x 表示输入结束,并将其倒序输出,代码如代码清单 7-12 所示。

代码清单 7-12

```
1  #include <iostream.h>           //包含输入/输出头文件
2  void main()
3  {
4      char a[10];                 //定义一维数组
5      int i,j;                    //定义变量
6      cout<<"请输入字符: ";
7      for(i=0;i<10;i++)           //循环接收用户输入
8      {
9          cin>>a[i];              //输入
10         if (a[i]=='x')            //输入字符 x 后结束输入
11             break;              //跳出循环
12     }
13     cout<<endl;
14     for(j=i-1;j>=0;j--)          //倒序输出
15         cout<<a[j]<<" ";        //输出
16     cout<<endl;
17 }
```

【运行结果】在 Visual C++ 6.0 中执行,其结果如图 7-15 所示。

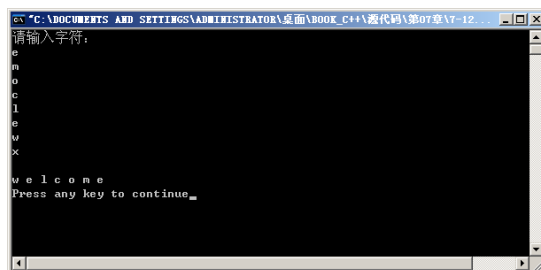


图 7-15 字符数组

【范例解析】上述代码将用户输入字符存储到字符数组中,并以变量 j 获取最后输入的字符数组元素下标(除 x 外,因此下标为 i-1),通过下标递减的方法实现倒序输出。



注意 字符数组与字符串是不一样的。两者最显著的区别在于，字符串的长度是其中字符的数目再加1，因为其包含了结束符'\0'，而字符数组的长度就是其中字符的数目。

【范例 7-13】字符串与字符数组的区别。该范例定义了一个字符串和一个字符数组，分别计算其长度，可看到字符串和字符数组的区别，实现代码如代码清单 7-13 所示。

代码清单 7-13

```

1  #include <iostream.h>                                //包含输入/输出头文件
2  void main()
3  {
4      char a[]={"Hello"};                               //声明一个字符串
5      char b[]={'H','e','l','l','o'};                   //声明一个字符数组
6      int la,lb;
7      la=sizeof(a)/sizeof(char);                         //计算字符串长度
8      lb=sizeof(b)/sizeof(char);                         //计算字符数组长度
9      cout<<"The length of a is : "<<la<<endl;        //输出结果
10     cout<<"The length of b is : "<<lb<<endl;
11 }

```

【运行结果】将上述代码在 Visual C++中执行，其结果如图 7-16 所示。

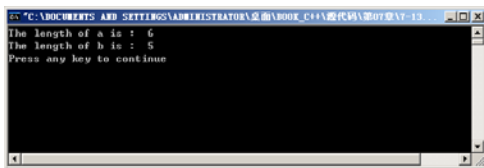


图 7-16 字符串与字符数组

【范例解析】以上代码中，声明了一个字符串 a 和一个字符数组 b[]，通过 sizeof()函数来计算字符数组的长度。需要读者注意的是上述求数组长度的函数 sizeof()，在实际应用中，应将字符数组的长度除以本机一个字符所占的长度，这是因为具体的机器其字符型数据类型所占长度不一样。

比如，sizeof(a)表示求数组 a 在内存中所占字节数，sizeof(char)表示求字符型数据在内存中所占字节数。使用表达式 sizeof(a)/sizeof(char)，可以使数组大小计算在 16 位机器和 32 位机器之间移植。因此，上述计算字符数组的长度所用的表达式为 sizeof(a)/sizeof(char)和 sizeof(b)/sizeof(char)。

7.5 数组与函数

在实际的应用中，数组经常作为函数参数，将数组中数据传送到另一个函数中。一般来说，传递可以采用两种方法。

- 数组元素作为函数的参数：当把数组元素作为函数的实参时，它的用法与普通变量作参数相同。将数组元素的值传送给形参进行函数体调用，函数调用完返回后，数组元素的值不变。这种传送方式是“值传送”方式，即只能从实参传送给形参，而不能从形参传送给实参。



注意 实参与形参的类型要相同。

- 数组名作为函数的参数：当用数组名作为函数的实参和形参时，传递的是数组的地址。这时实参数组和形参数组应该分别在它们所在的函数中定义。此时采取的不是“值传送”方式，而是“地址传送”方式，即把实参数组的起始地址传送给形参数组，这样形参数

组就和实参数组共占同一段内存单元,当形参值发生变化时,实参值也发生变化。在将数组作为函数参数进行传递时,读者需要注意如下事项:

- 实参数组与形参数组类型要一致。
- 形参数组的长度不要超过实参数组的长度。实参数组必须定义为具有确定长度的数组,而形参数组可以不定义长度,只在数组名后加一个空的方括号,同时在被调用的函数中另设一个参数用来传递元素的个数。
- 可以在被调用函数中采用降维处理,即用单重循环来遍历二维数组中的所有元素。此时调用函数中的数组不要用数组名表示,而要用第一个元素的地址表示。

【范例 7-14】数组作为函数参数进行传递。该范例实现输入 10 个学生的成绩,求出平均成绩,并将低于平均成绩的分打印出来,实现代码如代码清单 7-14 所示。

代码清单 7-14

```

1  #include <iostream.h>                //包含输入/输出头文件
2  #include <iomanip.h>
3  void readdata(float score[10])        //定义函数接收用户输入
4  {
5      cout<<"Please input 10 student's score:"<<endl;
6      for(int i=0;i<10;i++)            //循环与输入语句结合
7          cin>>score[i];
8      return;
9  }
10 float aver(float score[10])           //定义函数求平均值
11 {
12     float sum=0;                      //定义变量并初始化
13     for(int i=0;i<10;i++)
14         sum+=score[i];                //求和
15     return(sum/10);                  //求平均值
16 }
17 void print(float score[10],float ave)  //定义函数输出低于平均分的分数
18 {
19     int i;
20     cout<<"the scores which are below the average:";
21     for(i=0;i<10;i++)                //循环语句
22         if(score[i]<ave)               //输出数组元素中所有大于平均值的元素
23             cout<<score[i]<<" ";
24     return;
25 }
26 void main()
27 {
28     void readdata(float score[10]);    //声明函数
29     float aver(float score[10]);       //声明函数
30     void print(float score[10],float ave); //声明函数
31     float ave,score[10];              //定义变量和数组
32     addata(score);                    //调用函数
33     ave=aver(score);                  //调用求平均值函数
34     cout<<"average="<<ave<<endl;      //输出平均值
35     print(score,ave);                 //调用输出低于平均分的分数的函数
36     cout<<endl;
37 }

```

【运行结果】该程序在 Visual C++ 中的运行结果如图 7-17 所示。

【范例解析】上述程序中,定义了三个函数,分别实现接收用户的输入函数 readdata,求平均分函数 aver 和输出低于平均分函数 print,其函数参数都包含一个存储分数的数组 score。

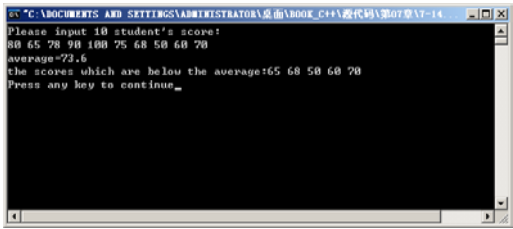


图 7-17 数组作函数参数

7.6 数组应用

在实际的应用中，数组的使用是很频繁的。由于实际生活中，经常需要处理相同类型的一类事物，这就需要使用到数组。在程序设计中，典型的数组应用有数组元素的查找、数组的排序等。本节将介绍这两种典型的应用如何使用数组实现。

7.6.1 顺序查找

在实际的应用程序中，经常要从数组中找出某个元素，获得其在数组中的位置，这就是查找问题。用数组来实现元素的查找主要有两种方式：顺序查找和折半查找。所谓顺序查找就是在一个已知无序队列中找出与给定关键字相同的数的具体位置。顺序查找的原理是让关键字与队列中的数从第一个开始逐个比较，直到找出与给定关键字相同的数为止，顺序查找的实现流程如图 7-18 所示。

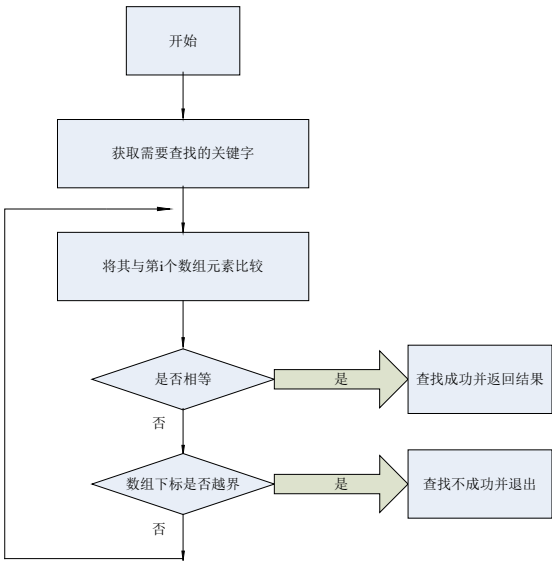


图 7-18 顺序查找

【范例 7-15】顺序查找的实现。该范例从键盘接收一个输入，从数组 a 中找出与该输入相同的元素值，并将该元素所在数组的位置输出，如没有找到则给出相关提示信息，实现代码如代码清单 7-15 所示。

代码清单 7-15

```
1  #include <iostream.h>
2  int main()
3  {
4      int a[15]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};    //定义数组并初始化
```

```

5      cout<<"Please input the number:"<<endl;
6      int iNum;                                //定义变量
7      cin>>iNum;                                //输入要查找的数值
8      for(int i=0; i<15; i++)                  //循环依次查找
9      {
10         if(a[i] == iNum)                      //找到
11         {
12             cout <<"The location is : " <<i+1 <<endl;
13             break;                            //找到后退出循环
14         }
15     }
16     if(i == 15)                                //没有找到
17     {
18         cout <<"It's not in the array" <<endl; //输出错误提示
19     }
20     return 0;
21 }

```

【运行结果】在 Visual C++ 中运行上述程序，输入目标值 6，则其运行结果如图 7-19 所示。

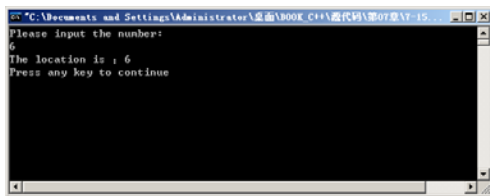


图 7-19 顺序查找执行结果

【范例解析】上述程序中，定义了一个含有 15 个元素的整型数组，从键盘接收需要查找的值，运用 for 循环语句依次将每个数组元素与该输入值比较，如相同则找到，将该数组元素所在的位置输出，并退出循环，结束程序，如没有找到则输出提示信息。



提示 顺序查找不需要数组中的元素有序，在最坏的情况下，顺序查找将搜索整个数组。该算法的效率较低。

读者可以看出，该程序使用到了循环语句，并在循环中含有一个条件语句，用来判断是否符合表达式中的条件，其程序执行流程如图 7-20 所示。

7.6.2 折半查找

折半查找又称二分法查找，与顺序查找相比较，其是一种效率较高的查找方法。需要注意的是，折半查找是有前提条件的，其要求数组中的元素是顺序排列的，即数组元素须按值升序排列或降序排列。如果没有这一前提，是不能进行折半查找的。

此处假定数组是有序的，那么折半查找的思想是：从初始的查找数组 $R[1...n]$ 开始，每次与当前查找区间的中间点位置上的元素值进行比较，相同则查找成功，不成功则当前的查找区间就缩小一半。这一过程不断重复直至找到目标值在数组中的位置，或者直至当前的查找区间为空（即查找失败）时为止，流程图如图 7-21 所示。

【范例 7-16】折半查找的实现。该范例实现与上述代码清单 7-15 相同的功能，查找某一用户指定元素所在数组的位置并输出，如没有找到则给出相关提示信息，实现代码如代码清单 7-16 所示。

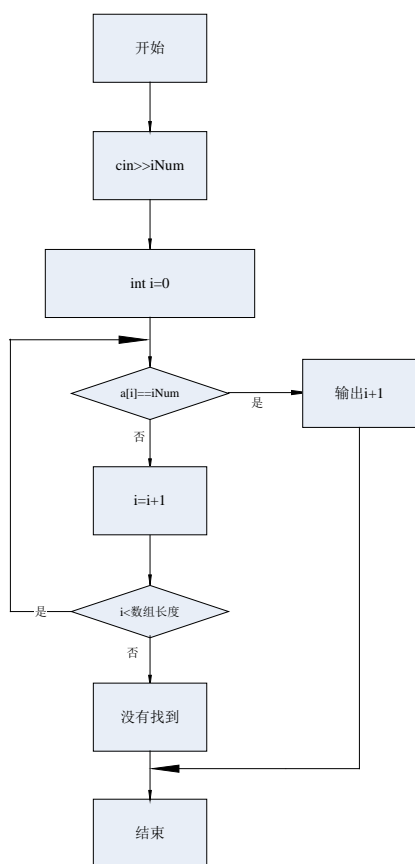


图 7-20 执行流程

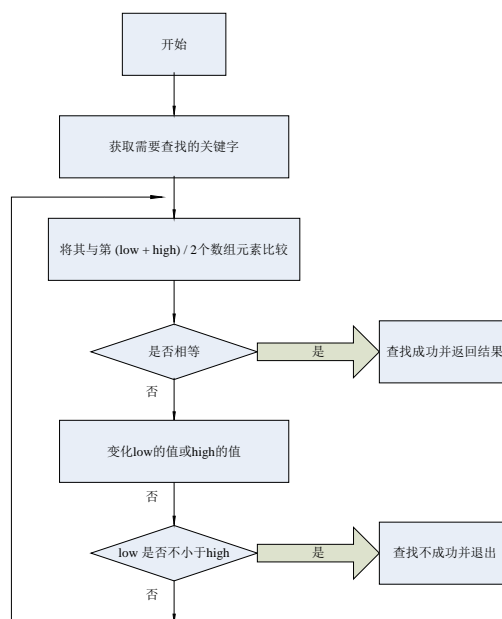


图 7-21 折半查找

代码清单 7-16

```

1  #include <iostream.h>
2  void main()
3  {
4      int a[15]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}; //定义数组并初始化
5      cout<<"Please input the number:"<<endl;
6      int key;
7      cin>>key; //接收目标数值
8      int low = 0; //定义变量并赋初值
9      int high = 14;
10     while (low <= high)
11     {
12         int mid = (low + high) / 2; //计算中间节点位置
13         if (key == a[mid]) //找到
14         {
15             cout<<"The location is : " <<mid+1<<endl; //返回找到的索引值
16             break;
17         }
18         else //没有找到
19         {
20             if (key < a[mid]) //在中间节点左侧查找
21                 high = mid - 1;
22             else //在中间节点右侧查找
23                 low = mid + 1;
24         }
25     }
26 }
  
```

```

25     }
26     if (low >= high)                                //没有找到
27         cout <<"It's not in the array" <<endl;      //输出提示信息
28 }

```

【运行结果】在 Visual C++ 中运行上述程序，运行该程序后输入需要查找的目标值为 16，则其运行结果如图 7-22 所示。

【范例解析】上述程序一开始就定义了变量 `low` 和 `high`，并给其赋初值为 0 和 14，表示数组的第一个元素和最后一个元素，接下来用循环依次判断其数组的中间节点是否与指定值相等，相等则输出位置，否则改变 `low` 的值或 `high` 的值，再进行比较，直到找出节点或 `low` 的值大于或等于 `high` 的值，即没有中间节点时则退出。

提 折半查找的算法执行效率较高，在最好的情况下，其只需要比较一次即可，最坏情况下则查找完整个数组，该算法要求数组基本有序。

上述程序中同样在循环语句中使用了条件判断语句，这是在实际的程序中较为常用的，该程序的执行流程如图 7-23 所示。

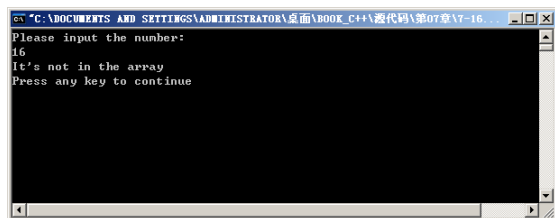


图 7-22 折半查找执行结果

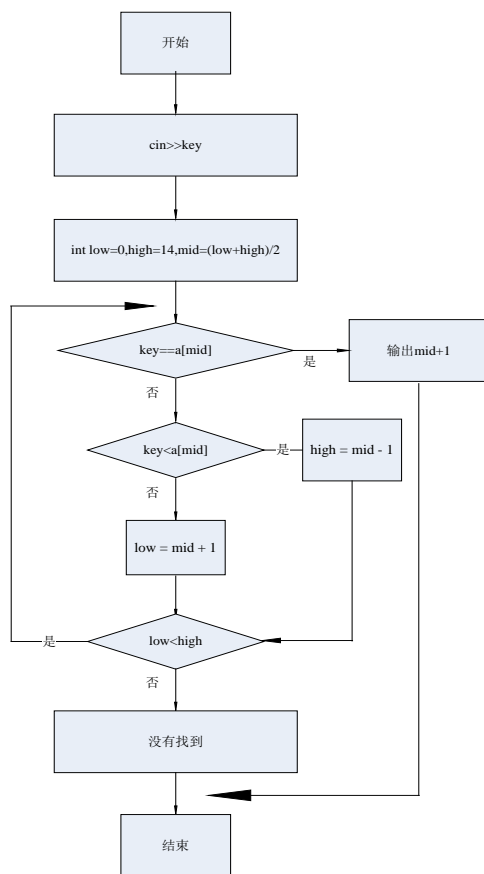


图 7-23 执行流程

提 读者可以看出，在数组基本有序的情况下，用折半查找不需要依次比较数组的每个元素，这大大提高了查找的效率。

7.6.3 排序

与查找相似的，排序也是实际程序中的一项基本应用。由于实际工作中处理的数量巨大，所以排序对算法本身的速度要求很高。由于篇幅限制，就不展开讲解各种排序算法，只简单介绍最常见的冒泡排序算法的 C++实现。

冒泡排序是一种最为常见也最原始的排序方法，其实现思想是依次两两比较待排序的数组元素；若为逆序（递增或递减）则进行交换，将待排序元素从左至右比较一遍称为一趟“冒泡”。每趟冒泡都将待排序列中的最大关键字交换到最后（或最前）位置，直到全部元素有序为止。例如，有一个整型数组 `int a[8]= {44,55,22,33,99,11,66,77}`，该数组共有 8 个记录。将其采用冒泡排序进行按由小到大的顺序排序，其执行如图 7-24 所示。

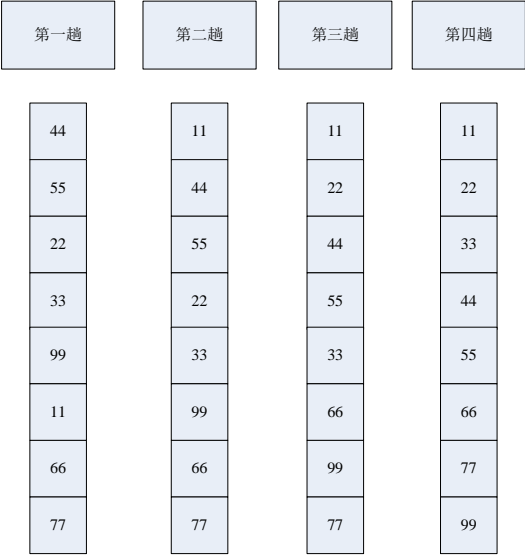


图 7-24 冒泡排序

【范例 7-17】冒泡排序的实现。代码清单 7-17 实现了上述整型数组 `a[8]`中数组元素的降序排列。

代码清单 7-17

```
1  #include <iostream.h>
2  void main()
3  {
4      int a[8]={44,55,22,33,99,11,66,77};           //定义数组并初始化
5      int iTemp;                                     //定义变量
6      cout<<"Before Sort: "<<endl;
7      for (int i=0;i<8;i++)                          //输出未排序前的数组
8          cout<<a[i]<<"\t";
9      for(i=1;i<8;i++)                                //双重循环
10     {
11         for(int j=7;j>=i;j--)
12         {
13             if(a[j]<a[j-1])                          //前一个元素大于后一个元素时交换
14             {
15                 iTemp = a[j-1];                      //交换两个元素
16                 a[j-1] = a[j];
17                 a[j] = iTemp;
18             }
19         }
20     }
```

```

19     }
20 }
21 cout<<endl<<"After Sort: "<<endl;           //输出提示
22 for(i=0;i<8;i++)                             //输出排序后的数组
23     cout<<a[i]<<"\t";
24     cout<<endl;                               //输出换行
25 }

```

【运行结果】在 Visual C++ 中执行上述程序，其返回结果如图 7-25 所示。

【范例解析】从上述程序读者可以看出，其使用了一个双重循环，内循环用于交换相邻的数组元素，外循环用于控制比较的次数。一般说来，冒泡排序的执行流程如图 7-26 所示。

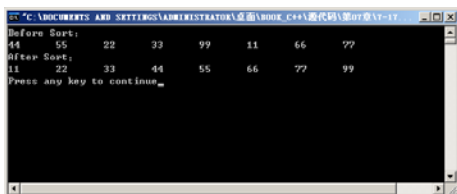


图 7-25 冒泡排序执行结果

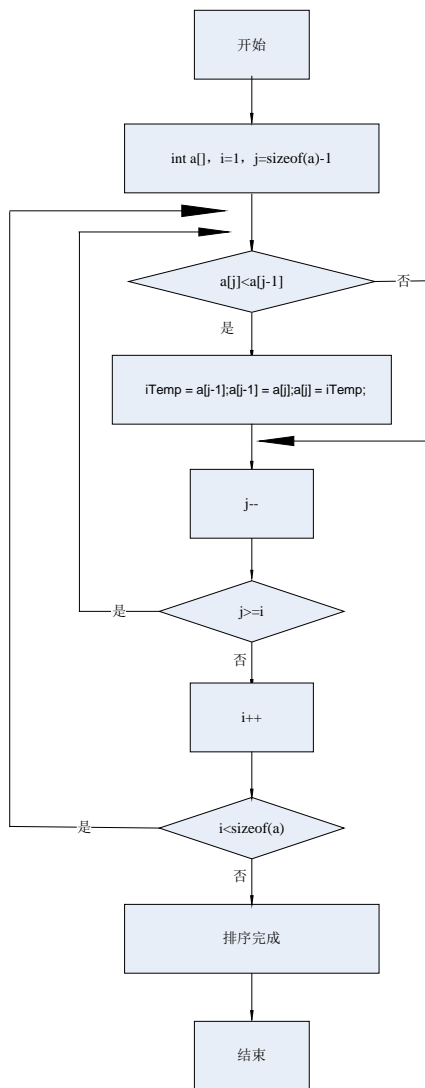


图 7-26 执行流程



提示 上述执行流程图中，sizeof()函数表示数组的长度。读者可以看到，当符合条件时将相邻两个数组元素相互交换，一直到扫描完成，这就是冒泡排序法。



7.7 小结

本章主要介绍了 C++ 中非常重要的一种存储类型——数组的相关内容，主要包括数组的声明、引用和赋值。其中，数组包含一维数组和多维数组。对于多维数组，本章重点讲解了较为常用的二维数组的声明和引用数组元素等内容。此外，字符数组和字符串也是实际中应用较多的，本章也做了简要介绍。对于数组作为函数的参数，将数组与函数结合起来，本章通过一个示例具体讲解了其应用。最后对数组在实际程序中的两种应用做了讲解，主要包括利用数组进行查找和排序的实现。

7.8 习题

1. 设有二维数组 `b` 和 `c`，在声明的同时进行了初始化，如下所示。

```
int b[2][3]={{1,2,3},{4,5,6}};  
int c[2][3]={1,2,3,4,5,6}
```

则 `b[1][1]` 和 `c[1][1]` 的值分别为多少？

【解答】该习题主要考查二维数组的初始化问题。在二维数组中，允许部分初始化。在对其中元素进行部分初始化时，不足部分补 0。因此，上述程序中元素 `b[1][1]` 获取的是二维数组第 2 行第 2 列的元素，即 5；`c[1][1]` 同样获取二维数组第 2 行第 2 列的元素，即 5。

2. 编写一个 C++ 程序，通过数组接收用户输入的 5 位同学的成绩，计算其平均成绩。

【解答】该习题主要考查数组元素的运算。此处先定义一个包含 5 个元素的浮点型数组，通过循环语句接收用户的输入，同时将这几个数组元素的总和求出。在退出循环后将总和与 5 进行除法运算，得出平均成绩并输出。其简要的实现代码如下所示。

```
for(i=0,ever;i<5;i++)  
{  
    cin>>a[i];  
    ever += a[i];  
}  
ever /= 5;  
cout<<"平均成绩是"<<ever<<endl;
```

3. 编写一个 C++ 程序，要求通过数组的形式接收用户输入的 10 个整数，找出其中的最大值并显示出来。例如，在用户屏幕输入 1 2 3 4 5 6 7 8 9 10 等 10 个数字，其返回值如图 7-27 所示。

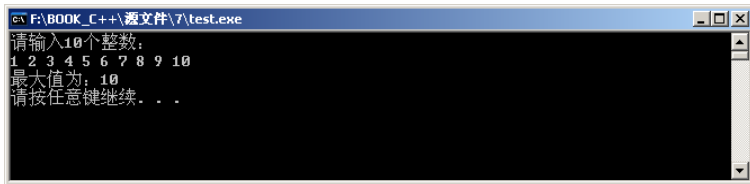


图 7-27 找出最大值

【解答】该习题主要考查数组元素的输入。首先必须定义一个包含 10 个元素的整型数组，通过一个循环将用户输入的 10 个数字全部存储到对应的数组元素中，然后通过循环依次比较数组中的所有元素，找到其中最大者将其输出即可。其简要的实现代码如下所示。

```
for(i=0;i<10;i++)  
    cin>>a[i];  
for(i=1,max=a[0];i<10;i++)  
    if(a[i]>max)  
        max = a[i];  
cout<<"最大值为: "<<max<<endl;
```

4. 从键盘输入 10 个整数, 检查整数 85 是否包含在这些数据中, 如包含则统计其被输入了几次。例如, 输入 10 个整数 44 78 98 85 61 35 85 41 73 85, 输出结果如图 7-28 所示。

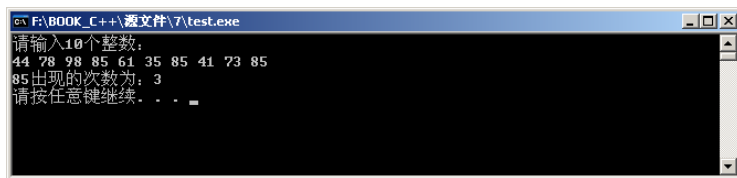


图 7-28 统计次数

【解答】该习题主要考查一维数组的元素匹配问题。首先定义一个包含 10 个元素的整型一维数组用于接收用户的输入, 同时判断用户的输入是否为 85, 如是则将个数累加 1, 一直到用户输入完成为止, 最后将个数输出。其简要的实现代码如下所示。

```
int num[10], sum;
sum = 0;
cout << "请输入 10 个整数: " << endl;
for (int i = 0; i < 10; i++)
{
    cin >> num[i];
    if (num[i] == 85)
        sum++;
}
cout << "85 出现的次数为: " << sum << endl;
```

5. 编写一个 C++ 程序, 从键盘上输入 10 个整型数字, 将其中重复的数字去掉, 并将剩余数字按照从小到大的顺序输出。例如, 输入 10 个整型数值, 如: 4 3 6 3 8 4 5 7 9 6, 其返回的输出结果如图 7-29 所示。

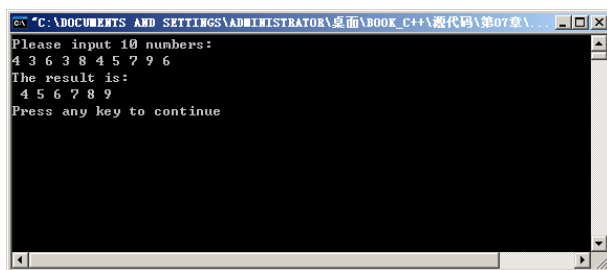


图 7-29 返回结果

【解答】该程序段首先需声明一个 sort() 函数, 其用于实现数组的冒泡排序。在主函数 main() 中, 接收键盘输入的 10 个数字, 存入数组中。执行 sort() 函数, 使得数组中的元素按值升序排列。最后通过一个 for 循环, 比较数组中相邻的两个元素, 如果这两个元素相等, 则不输出重复元素。其简要实现的代码如下所示。

```
#include <iostream.h>
void sort(int a[], int n); // 声明函数 sort
int main()
{
    int i, a[10]; // 定义整型变量和数组
    cout << "Please input 10 numbers: " << endl; // 输入提示
    for (i = 0; i < 10; i++) // 给数组赋初值
        cin >> a[i]; // 接收从键盘的输入
    sort(a, 10); // 冒泡排序
    cout << "The result is: " << endl;
```



```
        for (i=1;i<10;i++)
            if (a[i]!=a[i-1])
                cout<<" "<<a[i];
            cout<<endl;
        return 0;
    }
    void sort(int a[],int n)
    {
        int i,j,t;
        for (i=0;i<n;i++)
            for (j=i+1;j<n;j++)
                if (a[i]>a[j])
                {
                    t=a[i];
                    a[i]=a[j];
                    a[j]=t;
                }
    }
```

//扫描数组
//如两两重复则不输出，否则输出
//输出元素，中间以一个空格隔开
//输出换行

//冒泡排序

//定义整型变量
//进入循环

//前一个元素大于后一个元素

//交换两个元素

//交换完成

第 8 章 指针

指针是 C++ 语言中的一个重要概念。由于使用指针可使程序简洁、紧凑和高效，所以对于每一个使用 C++ 语言的人，都应该掌握指针的使用方法。以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为一个学习的参照。

- 了解指针的概念。
- 熟练掌握指针的定义和运算。
- 掌握指针与数组、函数和字符串的运算。
- 掌握指向指针的指针的使用。

8.1 指针概述

准确地理解指针的概念是掌握指针最重要的一个步骤，许多初学者不能掌握指针的使用方法都是由于没有正确理解指针的概念。

8.1.1 指针是什么

简单来说，指针是一个地址，其指向存储某一个数据的存储地址。此外，还有一个指针变量的概念，指针变量是一种特殊性质的变量。指针变量是把地址存放在一个变量中，然后通过先找出地址变量中的值（一个地址），再由此地址找到最终要访问的变量的方法，这就是指针变量及其访问方法，而地址变量就是指针。

对于指针我们可以这样理解，比如一个人要到某地去，但不认识路，于是去问交警。然后交警把该地方的地址写在了一张纸上，并且给了该问路人。那么交警写的地址就是指针，指向要去的地址，而那张纸就是指针变量，用于存储指针。例如，在内存中存储了一个变量 `a`，其值为 5，那么通过指针访问该变量如图 8-1 所示。

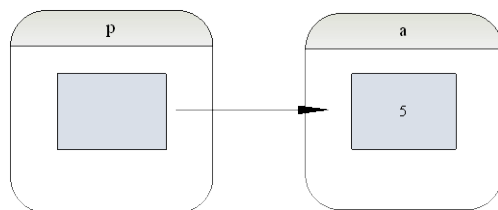


图 8-1 指针

由图 8-1 可以看到，指针变量 `p` 指向变量 `a`。在理解“指向”的时候，应该了解它指的是：`p` 中存有 `a` 的地址，通过该地址就能找到 `a`。因此，在 C++ 语言中用指针来表示指向关系，即指针就是地址。一个变量的指针就是该变量的地址。存放地址的变量，就是指针变量。如果在程序中定义了一个变量，在编译时就会给这个变量分配内存单元。系统根据程序中定义的变量类型，分配一定长度的内存空间，每个内存单元中存放着变量的值。指针除了可以指向变量之外，还可以指向内存中其他任何数据结构，如数组、结构体和联合体等，它还可以指向函数，这些将在后面章节陆续介绍。应该注意，在程序中参加数据处理的量不是指针本身的量，因为指针本身是个地址量。而指针所指向的变量，即指针所指向的内存区域中的数据（称为指针的



目标)才是需要处理的数据。这就是 C++ 语言中利用指针处理数据的特点。

8.1.2 定义指针

前面提到了,指针是一个变量,在程序中使用时,必须先声明,后使用。在指针声明的同时也可以进行初始化。指针的定义指出了指针的存储类型和数据类型,定义的语法形式如下:

存储类型名 数据类型 *指针变量名

其中各部分的作用如下。

- 存储类型名: C++ 中的存储类型一般有静态存储、栈和自动类型三种,一般的默认值为自动类型 `auto`。
- 数据类型: 可以是任一基本类型名、派生类型名及自定义类型名;也可以是由 `<类型名>*` 表示的指针类型名,称它为多级指针。

例如,下面定义了名为 `p1`、`p2` 和 `p3` 的三个不同类型指针。

```
int *p1;
static int *p2;
char *p3;
```



注意 把指针指向的变量的数据类型称为指针的数据类型;而任何一个指针变量本身数据值的类型都是 `unsigned long int`。

上述定义中,在指针变量名前的符号“*”表示指向运算。指针变量的类型确定后只能指向这种既定的数据类型,不可指向其他类型的数据。需要注意的是,定义一个指针变量必须用符号“*”,它表明其后的变量是指针变量,但不要认为“*p”是指针变量,指针变量是 `p` 而不是 `*p`。

此外,有相同存储类型和数据类型的指针可以在一行中说明,它们也可以和同类型的普通变量在一起说明。例如:

```
int *p1, *p2, *p3;
char m, *da;
```

在指针定义的例子中,第一行声明了三个 `int` 型指针 `p1`, `p2`, `p3`;第二行声明了一个 `char` 型变量 `m` 和一个指针 `da`,这是允许的。但是,当在一行中定义多个同一类型的指针时,用逗号隔开各指针变量标识符,并且每个变量前都要加上“*”。

8.1.3 指针的初始化

定义了一个指针后,在使用此指针前,必须首先给它赋一个合法的值。否则,程序中对指针的使用就有可能导致系统崩溃。可以在定义指针的同时通过初始化来给指针赋值,也可以在使用之前给指针赋值。下面首先来了解指针的初始化。由于指针是保持地址的变量,所以初始化时赋予它的初值必须是地址量。指针初始化的一般形式如下:

存储类型 数据类型 *指针名=初始地址值;

例如,下面语句将变量 `a` 的内存地址作为初始值赋予 `int` 型指针 `pa`。

```
int a,*pa=&a;
```

这种写法与下面的写法是等价的:

```
int a;
int *pa=&a;
```

当把一个变量的内存地址作为初始值赋给指针时,该变量必须在指针初始化之前已作说明。其道理很简单,变量只有在说明之后才被分配一定的内存地址。此外,该变量的数据类型

必须与指针的数据类型一致。下面的例子是把一个指针初始化为空指针。

```
int *px=0;
```

这个语句将指针 `px` 的值初始化为 0。值为 0 的指针叫做空指针。为了使用安全起见，一般来说，在定义指针时，最好初始化，哪怕是初始化为空指针。



提示 如果在定义指针时，指针初始化为 0 或者根本没有初始化。那么在使用此指针前，就必须给它赋予有意义的值。例如：

```
int n,*p1;    //定义指针 p1 时没有初始化
p1=&n;        //给指针 p1 赋值为 int 型变量 n 的地址
```

或者

```
int n,*p1=0;  //定义指针 p1 时初始化为 0
p1=&n;        //给指针 p1 赋值为 int 型变量 n 的地址
```

上述的初始化都是正确的。同时，也可以向一个指针赋初值作为另一个指针变量，即把另一个已经初始化的指针赋予一个指针。此时，这两个指针指向同一变量的内存地址，如下所示：

```
int n;
int *p1=&n;    //指针 p1 的值初始化为变量 n 的地址
int *p2=p1;    //指针 p2 的值初始化为指针变量 p1
```

这种写法与下面的写法是等价的：

```
int n;
int *p1=&n;    //指针 p1 的值初始化为变量 n 的地址
int *p2=&n;    //指针 p2 的值初始化为变量 n 的地址
```

在利用指针访问变量的值时，也可以通过指针给变量间接赋值。

【范例 8-1】定义指针并初始化。该范例定义了一个指针变量 `pa`，并为其赋初值，其实现代码如代码清单 8-1 所示。

代码清单 8-1

```
1  #include <iostream.h>
2  void main()
3  {
4      int a=1;                //定义整型变量并初始化
5      cout<<"a="<<"\t"<<a<<endl;    //输出变量 a 的值
6      int *pa=&a;              //定义指针 pa 并初始化
7      *pa=2;                  //给 *pa 赋值，即给 a 赋值
8      cout<<"a="<<"\t"<<a<<endl;    //输出变量 a 的值
9  }
```

【运行结果】在 Visual C++ 中新建一个 **【C++ Source File】**，输入上述代码，编译无误后执行，其结果如图 8-2 所示。

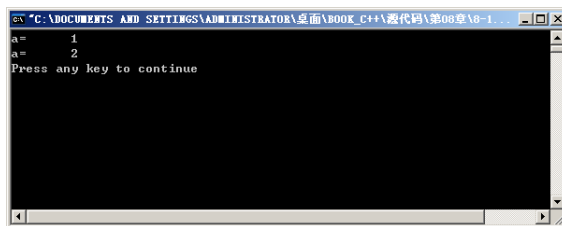


图 8-2 指针的初始化



【范例解析】上述程序中，首先定义变量 `a`，并为其赋初值 1，此时输出 `a` 的值为 1，然后定义指针变量 `pa`，其指向变量 `a`，即其中存储了 `a` 的存储地址，使用赋值语句 “`*pa=2;`” 相当于语句 “`a=2`”，因此其后 `a` 的值变为 2。

8.2 指针的运算

8.1 节介绍了指针的概念和定义，下面将重点介绍在具体的程序中如何使用指针，以及使用指针所带来的好处。指针运算是以指针变量所持有的地址值为运算量进行的运算。因此，指针运算的实质是地址的计算。由于指针是持有地址量的变量这一特性，指针的运算与普通变量的运算在种类上和意义上都是不同的。指针运算的种类是有限的，它只能进行取地址和取值运算、算术运算、关系运算和赋值运算。

8.2.1 取地址与取值运算

如果说明了一个指针，并使其值为某个变量的地址，则可以通过这个指针间接地访问在这个地址中存储的值。经过上面部分的学习我们知道，在 C++ 语言中有两个有关指针的特别运算符。

- 运算符 `&`：为取地址运算符，`&x` 的值为 `x` 的地址。
- 运算符 `*`：指针运算符，或指向运算符，也称间接运算符，`*p` 代表 `p` 所指向的变量。

由此可看出，利用指针来访问变量值需要使用间接访问运算符 “`*`”。需要注意的是，在指针变量的定义和指针变量的引用中都有 `*p`。但引用指针时的 `*p` 与定义指针变量时用的 `*p` 是有区别的，它们形式上有些相似，而含义是不同的。例如：

```
int a=1,*pa=&a;
cout<<*pa;
```



提示 用 `cout` 语句输出的是变量 `a` 的值 1，即 `*pa` 就是代表变量 `a`。上面的 “`cout<<*pa;`” 语句与 “`cout<<a;`” 语句的作用相同。

由于引进了指针的概念，读者在程序中要注意区分下面三种表示方法所具有的不同意义。例如，有一个指针 `px`，其不同格式代表的意义如下：

- `px`——指针变量，它的内容是地址量。
- `*px`——指针的目标变量，它的内容是数据。
- `&px`——指针变量占用的存储区域的地址。

【范例 8-2】指针的间接访问，取值和取地址操作。该范例定义了一个指针变量 `p`，其指向整型变量 `a`，取该指针的值、地址，依次输出其 `p`、`*p` 和 `&p` 的值，代码如代码清单 8-2 所示。

代码清单 8-2

```
1  #include <iostream.h>
2  void main()
3  {
4      int a=1;                //定义变量 a
5      int *p;                 //定义指针 p
6      p=&a;                    //初始化指针
7      cout<<"p="<<"\t"<<p<<endl;    //输出
8      cout<<"*p="<<"\t"<<*p<<endl;
9      cout<<"&p="<<"\t"<<&p<<endl;
10 }
```

【运行结果】在 Visual C++ 中运行上述程序，其运行结果如图 8-3 所示。

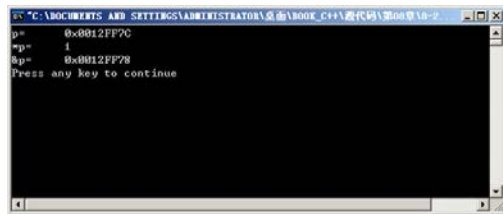


图 8-3 指针的间接访问

【范例解析】读者可以看到，上述返回值中，只有 `*p` 的值为 `a` 的值，其内容才是有意义的数据。而 `p` 的值为存储变量 `a` 的地址，`&p` 的值为指针变量 `p` 的存储地址，这些都是由计算机随机分配，没有实际意义。因此，在实际程序中，读者一定要清楚如何使用指针访问数据。



注意 在对指针的操作中，使用最多的是取值和取址。取值，采用运算符 `*` 实现；取地址，采用运算符 `&` 实现。

8.2.2 指针的算术运算

指针的算术运算是按 C++ 语言地址计算规则进行的，这种运算与指针指向的数据类型有密切关系，也就是 C++ 语言的地址计算与地址中存放的数据长度有关。设 `px` 和 `py` 是指向具有相同数据类型的一组若干数据的指针，`n` 是整数，则指针可以进行的算术运算有如下几种：

`px+n`, `px-n`, `px++`, `++px`

`px--`, `--px`, `px-py`

下面分别介绍这几种算术运算的实现。

1. 指针与整数的加减运算：(`px+n`, `px-n`)

指针作为地址量加上或减去一个整数 `n`，其意义是指针当前指向位置的前方或后方第 `n` 个数据的位置。由于指针可以指向不同数据类型，即数据长度不同的数据，所以这种运算的结果取决于指针指向的数据类型。

例如，假设有一个指向单字节字符类型变量的指针和另一个指向双字节整数类型的指针。当字符指针加 1 时，实际结果是指针中的地址值加 1；而整数指针加 1 时，实际结果是指针中的地址值加 2。因此，对于某种数据类型的指针 `p` 来说：

- `p+n` 的实际操作是：`(p)+n*sizeof(数据类型)`
- `p-n` 的实际操作是：`(p)-n*sizeof(数据类型)`

其中，`(p)` 表示指针 `p` 中的地址值，而不是 `&p`，`sizeof(数据类型)` 的长度单位为字节。

在 C++ 语言中，指针加减运算一般用在对数组元素进行操作的场合。通过对指向数组的指针进行加减运算，可以使指针指向数组中不同的元素。同时也必须注意越界问题。

2. 指针加 1、减 1 运算：(`px++`, `++px`, `px--`, `--px`)

指针加 1、减 1 单项运算也是地址计算，它具有上述的计算特点，指针的加 1、减 1 单项运算是指针中的地址值的变化。指针 `++` 运算后就指向了下一个数据的位置，`--` 运算后就指向了上一个数据的位置。运算后指针地址值的变化量取决于它指向的数据类型。

例如，一个 `int` 型指针 `p` 存放的地址为 1010，当执行 `p++` 后，`p` 存放的地址为 1012，即指针 `p` 指向了下一个数据的地址。指针加 1、减 1 单项运算也分为前置运算和后置运算，当它们



和其他运算出现在一个表达式中时，要注意它们之间的结合规则和运算顺序。例如：

```
y=*px++
```

该表达式中有三种运算：`=`、`*`和`++`。由前面介绍的运算符的优先级可知，`*`和`++`优先于`=`。`*`和`++`属于同级运算，其结合规则是从右至左。所以`++`运算是对 `px` 进行的。它相当于：

```
y=*(px++)
```

这里 `px++` 是后置运算。因此该表达式的运算顺序是：访问 `px` 当前值指向的目标，把目标变量的值赋予 `y`，然后 `px` 加 1 指向下一个目标。

例如，字符串复制函数 `strcpy(s, t)` 的实现就使用到了这一运算，`strcpy(s, t)` 是标准函数库中的一个函数，函数体中使用了指针后置运算：`(*s++=*t++)!='\0'`，其目的是把 `t` 的目标变量的值赋予 `s` 的目标变量，然后判断赋值表达式的结果值，即赋的值是否不等于 `'\0'`。`s` 和 `t` 的值使用后执行加 1 运算，分别指向下一个目标，函数中循环体是空语句。

3. 指针的相减运算：(px-py)

如果两个指针 `px` 和 `py` 所指向的变量类型相同，则可以对它们进行相减运算。`px-py` 运算的结果值是两指针指向的地址位置之间的数据个数。由此看出，两指针相减实质上也是地址计算。它执行的运算不是两指针持有的地址值相减，而是按下列公式得出结果。

$$((px)-(py))/\text{数据长度}$$


提示 上式中 `(px)` 和 `(py)` 分别表示指针 `px` 和 `py` 的地址值，所以，两指针相减的结果值不是地址量，而是一个整数。

指针的相减运算一般也用在数组进行的操作中，比如：

```
int x[5],a;
int *px=&x[1],*py=&x[4];
a=py-px;
```

这里的变量 `a` 就表示数组元素 `x[1]` 和 `x[4]` 之间相隔的元素个数。

【范例 8-3】 指针的算术运算。该范例定义了 2 个指向数组的指针，用以完成指针与整数的相加运算、指针的加 1 减 1 运算和指针之间的相减运算，实现代码如代码清单 8-3 所示。

代码清单 8-3

```
1  #include <iostream.h>
2  void main()
3  {
4      int a[5]={1,2,3,4,5};           //定义数组
5      int *pa,*pb;                     //定义指针
6      pa=&a[0];                         //指针赋初值
7      pb=&a[4];
8      cout<<"*pa="<<*pa<<endl;       //输出指针执行的值
9      cout<<"*pb="<<*pb<<endl;
10     pa=pa+2;                          //指针与整数的相加运算
11     cout<<"pa+1="<<*pa<<endl;
12     pa=pa++;                           //指针加 1 运算
13     cout<<"pa++="<<*pa<<endl;
14     int c=pb-pa;                       //指针的相减运算
15     cout<<"pb-pa="<<c<<endl;
16 }
```

【运行结果】 同样，在 Visual C++ 中运行上述程序，其结果如图 8-4 所示。

【范例解析】 读者可以看出，上述代码实现了指针与整数 2 的相加，结果为指针指向后两

个元素；实现了指针的加 1 运算，结果为指针指向下一个元素；实现了指针的相减运算，结果为两个指针所指元素相隔的个数。

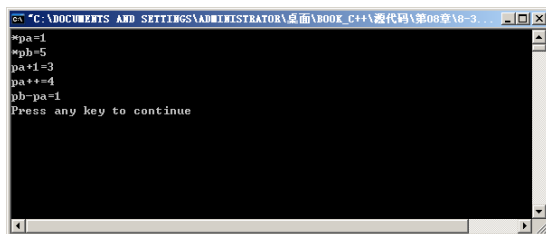


图 8-4 指针的算术运算

8.2.3 指针的关系运算

与指针的算术运算类似，在两个指向相同类型变量的指针之间可以进行各种关系运算。两指针之间的关系运算表示它们指向的地址位置之间的关系，例如：

```
int a;
int *p=&a,*q=p;
```

若上面声明的两个指针作 $p==q$ 运算，其结果为 1 (true)，即指针 p 、 q 指向同一个变量。两指针相等的概念是两指针指向同一位置。因此，假设数据在内存中的存储逻辑是由前向后，那么指向后方的指针大于指向前方的指针。也就是说，对于两指针 p 和 q 之间的关系表达式：

$p < q$

若 p 指向位置在 q 指向位置的前方，则该表达式的结果值为 1，反之为 0。指向不同数据类型的指针之间的关系运算没有意义，指针与非 0 整数之间的关系运算也是没有意义的。但是指针可以和零之间进行等于或不等于的关系运算，即：

$p == 0$

或

$p != 0$



提示

根据指针与零的比较，可以用于判断指针 p 是否为空指针。

【范例 8-4】 指针的关系运算。该范例实现两个指针之间的关系运算。当其运算的结果为真时，返回 1，反之结果为假时，返回 0，实现代码如代码清单 8-4 所示。

代码清单 8-4

```
1  #include <iostream.h>
2  void main()
3  {
4      int a[5]={1,2,3,4,5};           //定义数组 a
5      int *pa,*pb;                   //定义两个指针变量
6      pa=&a[0];                       //指针初始化，指向第一个数组元素
7      pb=&a[4];                       //指针初始化，指向最后一个数组元素
8      bool flag;                     //定义布尔型变量
9      flag=(pa>pb);                  //进行指针的关系运算
10     cout<<"pa>pb is "<<flag<<endl;  //输出结果
11     flag=(pa<pb);                  //进行指针的关系运算
12     cout<<"pa<pb is "<<flag<<endl;
13 }
```



【运行结果】在 Visual C++ 中运算上述代码，其结果如图 8-5 所示。

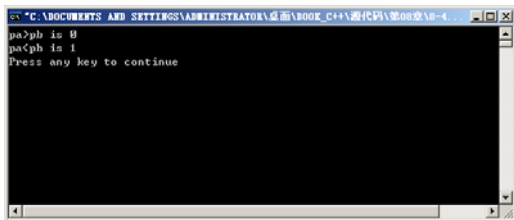


图 8-5 指针的关系运算

【范例解析】读者可以看出，指针之间可以进行关系运算。上述程序中，pa 指向的是第一个数组元素，pb 指向最后一个数组元素，因此，pb 的值大于 pa 的值。

8.2.4 指针的赋值运算

当向指针变量赋值时，赋的值必须是地址常量或变量，不能是普通整数。指针赋值运算常见的有以下几种形式。

- 把一个变量的地址赋予一个指向相同数据类型的指针，例如：

```
char a, *p;  
p=&a;
```

- 把一个指针的值赋予相同数据类型的另外一个指针，例如：

```
int *p, *q;  
p=q;
```

- 把数组的地址赋予指向相同数据类型的指针。例如：

```
char a[10], *pa;  
pa=a;
```

除了上述几种形式，前面提到过指针与整数的运算也是指针的赋值运算中的形式。因此，实际程序中还经常使用下列赋值运算：

```
int *p, *q, n;  
p=q+n;  
p=q-n;  
p+=n;  
p-=n;
```

这几种指针的赋值运算在前面的示例代码中基本都使用到了，此处就不再举例赘述，读者可自行参阅前面章节的示例代码。

8.2.5 void 指针和 const 指针

在 C++ 语言中，可以声明指向 void 类型的指针，指向 void 类型的指针称为 void 指针。此外，在声明指针时，还可以用关键字 const 进行修饰，用关键字 const 修饰的指针称为 const 指针。

1. void 指针

前面章节提到了，一般来说只能用指向相同类型的指针给另一个指针赋值，而在不同类型的指针之间进行赋值是错误的。比如：

```
int a,b;  
int *p1=&a,*p2=p1;           //正确  
int a;  
int *p1=&a;  
double *p2=p1;                //错误
```



注意 上述语句中的两个指针 p1、p2 指向的类型不同，因此，除非进行强制类型转换，否则它们之间不能相互赋值。

void 指针是一个特例。C++ 语言允许使用空类型 (void) 指针，即不指定指针指向一个固定的类型，其定义格式为：

```
void *p;
```

它表示指针变量 p 不指向一个确定的类型数据，它的作用仅仅是用来存放一个地址。void 指针它可以指向任何类型的 C++ 数据。也就是说，可以用任何类型的指针直接给 void 指针赋值。不过，如果需要将 void 指针的值赋给其他类型的指针，则需要进行强制类型转换。比如：

```
int a;
int *p1=&a;
void *p2=p1;
int *p4=(int *)p2;
```

2. const 指针

如果在指针定义前加上关键字 const，就包含一些特殊含义，而关键字 const 放在不同的位置表示的意义也不相同，主要如下：

- 关键字 const 放在指针类型前，就是声明一个指向常量的指针。此时，在程序中不能通过指针来改变它所指向的值，但是指针本身的值可以改变，即指针可以指向其他数据。
- 关键字 const 放在“*”号和指针名之间，就是声明一个指针常量（也称常指针）。因此，指针本身的值不可改变，也即它不能再指向其他数据，但它所指向的数据的值可以改变。
- 在指针类型前和“*”号和指针名之间都加关键字 const，则是声明一个指向常量的指针常量，指针本身的值不可改变，它所指向的数据的值也不能通过指针改变。

【范例 8-5】void 指针和 const 指针的应用。该范例定义了 void 指针和 const 指针，实现指针强制类型转换和指向常量的指针，读者可以仔细观察其定义方法和使用范畴，代码如代码清单 8-5 所示。

代码清单 8-5

```
1  #include <iostream.h>
2  void main()
3  {
4      int a=1;
5      int *p1=&a;                //定义指针并初始化
6      void *p2=p1;              //定义 void 指针并赋值
7      int *p3=(int *)p2;        //强制类型转换
8      cout<<"*p3= " <<*p3<<endl; //输出
9      const int *p4;            //定义 const 指针
20     int * const p5=&a;         //定义 const 指针并初始化
11     const int * const p6=&a;
12     cout<<"*p5= " <<*p5<<endl; //输出指针指向的值
13     cout<<"*p6= " <<*p6<<endl;
14 }
```

【运行结果】在 Visual C++ 中运行上述代码，其结果如图 8-6 所示。

【范例解析】读者可以看到，上述程序定义了 6 个指针变量，其中 p2 为 void 指针，而 p3 的值来源于对 p2 进行强制类型转换(int *)得来，p4、p5 和 p6 为三种不同形式的 const 指针。

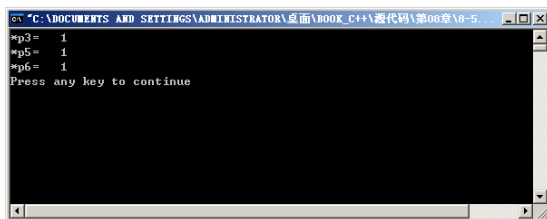


图 8-6 void 指针和 const 指针

8.3 指针与数组

前面已经提到了，指针在数组中使用较为频繁，代码示例中就定义过指向数组的指针，如代码 8-3 中定义的 `pa` 和 `pb` 指针就分别指向数组的第一个元素和最后一个元素。事实上，由于数组名表示的是该数组的首地址，所以如果定义一个指针指向数组，则可如下声明：

```
int a[10];
int *pa=a;
```

这个语句定义了一个指针 `pa`，并把 `pa` 初始化为指向数组 `int a[10]` 的指针，即指针 `pa` 指向数组的第一个元素。这时，不需要使用取地址运算符“&”。上述声明方式与下面的语句等价：

```
int a[10];
int *pa=&a[0];
```



注意 因为“&a[0]”表示数组 `a` 中第一个元素所在的内存地址值，而 `a` 也表示数组的第一个元素所在的内存地址值。

8.3.1 访问数组元素的方法

要访问或使用一个数组元素，可以用三种不同的方法：下标法、地址法，还有一种是指针法。

1. 下标法

下标法在前面已学习过，即指出数组名和下标值，系统就会找到该元素。数组用其下标变化实行对内存中的数组元素进行处理。例如，程序中说明了一个数组：

```
int a[5];
```

则编译系统在一定的内存区域为该数组分配了存放 `int` 型数据的 5 个连续存储空间，它们分别是 `a[0]`，`a[1]`，……`a[4]`，如图 8-7 所示。

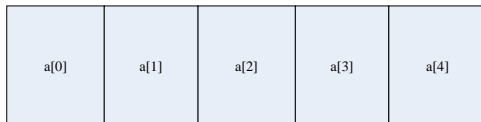


图 8-7 数组的内存空间

如果给定一个变量 `i`，其值区间为 `0~4`，那么 `a[i]` 就可以表示从数组存储首地址开始的第 `i` 个元素变量。在程序中通过 `i` 的变化就可以处理数组中的任何元素，`a[i]` 就是用下标法表示的数组元素。

2. 地址法

同样，前面已经介绍，一个数组名代表它的起始地址，而地址法即通过地址访问某一数组元素。以数组 `a[5]` 为例，程序中说明了以下数组：

```
int a[5];
```

则 a 的值就是数组的起始地址, 即 a 指向 $a[0]$, $a+1$ 指向 $a[1]$, ……同样, $a+i$ 是 $a[i]$ 的地址, 通过 $a+i$ 的地址可以找到 $a[i]$ 元素, 即 $*(a+i)$ 就是 $a[i]$ 。例如, 要访问数组元素 $a[3]$, 下面两种方式是等价的:

```
a[3]           //下标法
*(a+3)         //地址法
```

提示 从另一个角度来看, $a+i$ 和 $\&a[i]$ 是相等的, 都是 $a[i]$ 的地址。注意要区分 $a[i]$ 和 $\&a[i]$ 两者示的不同含义, $a[i]$ 是 a 数组第 i 个元素的值, 而 $\&a[i]$ 是 $a[i]$ 元素的地址。

3. 指针法

除上述两种方法之外, 还可以定义一个指针变量, 指向一数组元素, 称为指针法。例如, 若程序中声明数组 $a[5]$ 的同时说明了一个 int 型指针:

```
int a[5];
int *pa;
```

此时可通过指针赋值运算, 将 pa 指向数组的首地址:

```
pa=a;
```

或

```
pa=&a[0];
```

则指针 pa 就指向了数组 a 的首地址。这里指针的目标变量 $*pa$ 就是 $a[0]$ 。根据 8.2 节介绍的指针运算的原理, $*(pa+1)$ 就是 $a[1]$, $*(pa+2)$ 就是 $a[2]$ ……即 $*(pa+i)$ 就是 $a[i]$ 。

【范例 8-6】 访问数组的三种方法的实现。该范例分别使用了以上这三种方法对数组元素进行访问, 实现代码如代码清单 8-6 所示。

代码清单 8-6

```
1  #include <iostream.h>
2  void main()
3  {
4      int a[5]={1,2,3,4,5};           //定义数组并初始化
5      int *p=a;                       //定义指针并初始化
6      int i=1;
7      cout<<"a[i]= "<<a[i]<<endl;      //下标法
8      cout<<"*(a+i)= "<<*(a+i)<<endl;  //地址法
9      cout<<"*(p+i)= "<<*(p+i)<<endl;  //指针法
10 }
```

【运行结果】 在 Visual C++ 中指向上述程序, 其结果如图 8-8 所示。

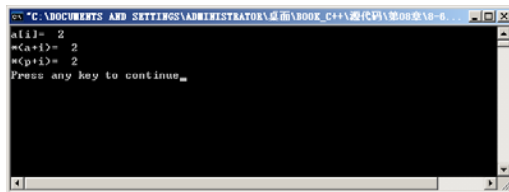


图 8-8 访问数组元素

【范例解析】 读者可以看出, 上述程序分别用下标法、地址法和指针法对数组的元素进行了访问。当然, 此处的变量 i 是既定的, 在实际程序编写中, 读者可以使用循环法使 i 的值变



化, 这样就能访问想要访问的数组元素了。

8.3.2 多维数组元素的访问

前面介绍的对数组元素的访问都是针对简单的一维数组, 事实上, 多维数组尤其是二维数组的访问也非常频繁, 那么对于多维数组, 又是如何用指针访问每个数组元素的呢? 首先以较为简单的二维数组为例来介绍其对数组元素的访问。例如, 下面语句定义了一个简单的二维数组 **a**:

```
int a[3][5];
```

由前面学习过的数组的内容, 读者可以知道, **a** 是以一个 3*5 的二维数组, 它有三行, 每一行都有其起始地址。

C++中, 以 **a[0]**、**a[1]**、**a[2]** 分别表示第 0 行、第 1 行、第 2 行的起始地址, 即该行第 0 列元素的地址。注意 **a[0]**、**a[1]**、**a[2]** 并不是一个元素, 而是一行首地址, 正如同一维数组名是数组起始地址一样, **a[0]** 的值等于 **&a[0][0]**, **a[1]** 的值等于 **&a[1][0]**, **a[2]** 的值等于 **&a[2][0]**。在同一行中类推方式是同样的, 比如: **a[0]+1** 的值等于 **&a[0][1]**, **a[1]+2** 的值等于 **&a[1][2]**, 等等。

因此, 对于二维数组中的元素 **a[i][j]** 有多种访问方法。以下是程序其中的一部分:

```
*(*(a+i)+j)
*(a[i]+j)
*(a+i)[j]
*(a+3*i+j)
```

如果用指针法访问二维数组中的元素, 处理的方法也有许多种。

【范例 8-7】 多维数组元素的访问实现。该范例定义了一个指向上述二维数组 **a** 首元素的指针 **p**, 并列出了通过这个指针访问数组元素 **a[i][j]** 的方法, 实现代码如代码清单 8-7 所示。

代码清单 8-7

```
1  #include <iostream.h>
2  void main()
3  {
4      int a[3][3]={1,2,3,4,5,6,7,8,9};           //定义二维数组
5      int i=1,j=1;                               //定义变量并初始化
6      int (*p)[3];                               //定义指向数组的数组指针
7      p=a;                                       //指针初始化
8      cout<<"*(*(a+i)+j)= "<<"*(*(a+i)+j)<<endl; //采用各种形式输出数组元素
9      cout<<"*(a[i]+j)= "<<"*(a[i]+j)<<endl;
10     cout<<"*(a+i)[j]= "<<"*(a+i)[j]<<endl;
11     cout<<"*(a+3*i+j)= "<<"*(a+3*i+j)<<endl;
12     cout<<"*(*(p+i)+j)= "<<"*(*(p+i)+j)<<endl;
13     cout<<"*(p[i]+j)= "<<"*(p[i]+j)<<endl;
14     cout<<"*(p+i)[j]= "<<"*(p+i)[j]<<endl;
15     cout<<"*(p+3*i+j)= "<<"*(p+3*i+j)<<endl;
16     cout<<"p[i][j]= "<<"p[i][j]<<endl;
17 }
```

【运行程序】 上述程序的执行结果如图 8-9 所示。

【范例解析】 上述程序中, 使用了多种方法访问二维数组中的元素。其中, 定义了一个数组指针 **p[3]**, 关于数组指针在后续章节中还将介绍。



注 读者可以看出, 上述程序中有一个地方发生越界了, 在程序中应避免这种问题。读者可以试着通过改变 **i** 和 **j** 的值来解决。

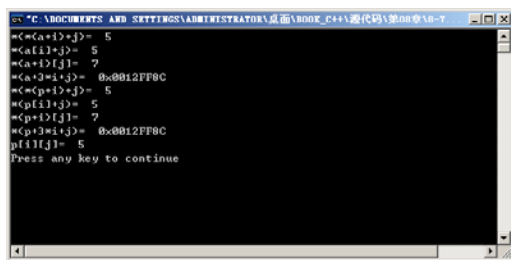


图 8-9 二维数组的访问

对于三维以上的多维数组，访问数组元素的方法原理上是一样的，只是在使用时一定要特别注意 C++ 中多维数组中各元素在内存单元中的存储顺序。在利用指针访问数组元素时，同样要注意越界问题。

8.3.3 数组指针与指针数组

在 C++ 语言中，数组指针就是一个指向数组的指针，代码 8-7 就使用到了数组指针，指针数组就是其元素为指针的数组。在学习 C++ 语言时，要注意对它们进行区分，不能等同起来。

1. 数组指针

数组指针是一个指向一维数组的指针变量，定义数组指针的格式为：

数据类型 (*指针名) [常量表达式];

例如，下列定义了一个指向包含 5 个整型元素的一维数组的指针。

```
int (*p)[5];
```

这个语句定义了一个数组指针 *p*，指向一个包含 5 个元素的一维数组，数组元素为整型。注意，**p* 两侧的圆括号不能省略，它表示 *p* 先与“*”结合，是指针变量。如果省略了圆括号，即写成 **p*[5] 的形式，由于方括号的优先级比星号高，则 *p* 先与方括号 [] 结合，是数组类型，那么语句 `int *p[5];` 是定义了一个指针数组。

2. 指针数组

指针数组就是其元素为指针的数组。它是指针的集合，它的每一个元素都是指针变量，并且它们具有相同的存储类型和指向相同的数据类型。说明指针数组的语法格式为：

数据类型 *指针数组名 [常量表达式];

其中，数据类型是指数组中各元素指针所指向的类型，同一指针数组中各指针元素指向的类型相同；指针数组名即为数组的首地址，是一个标识符；常量表达式指出这个数组中的元素个数。

例如，下面定义了几个指针数组：

```
int *p1[6];
float *p2[3][4];
```



提示 具有相同类型的指针数组可以在一起说明，它们也可以与变量，指针等一起说明。例如：

```
int a, *p[2];
```

指针数组在使用前必须首先赋值，也可以利用初始化赋值。一般来说，指针数组主要用于字符串的操作，例如：

```
static char *name[5] = {"Tom", "John", "Mary", "Smith Black", "Rose"};
```




其中 `name` 是一维数组，每一个元素都是指向字符数据的指针类型数据，其中 `name[0]` 指向第一个字符串“Tom”，`name[1]` 指向第二个字符串“John”，……。用指针数组处理字符串不仅可以节省内存，而且还可以提高运算效率。例如，想对 5 个姓名排序，将字符串交换位置速度慢，而交换地址则速度快得多。

【范例 8-8】 数组指针和指针数组的应用。该范例定义了一个数组指针和一个指针数组，并对数组中的元素进行访问。注意其相互之间的不同，其实现代码如代码清单 8-8 所示。

代码清单 8-8

```

1  #include <iostream.h>
2  void main()
3  {
4      int a[5]={1,3,5,7,9};           //定义数组
5      int (*pa)[5]=&a;               //定义数组指针 pa 并初始化
6      cout<<"*pa[0]= "<<*pa[0]<<endl; //访问数组元素
7      cout<<"*(pa[0]+1)= "<<*(pa[0]+1)<<endl;
8      char *pb[5]= {"Tom", "John", "Mary", "Smith Black", "Rose"};
                                           //定义指针数组 pb 并初始化
9      cout<<"pb[0]= "<<pb[0]<<endl;   //访问数组元素
10     cout<<"pb[1]= "<<pb[1]<<endl;
11 }

```

【运行结果】 在 Visual C++ 中执行上述代码，其结果如图 8-10 所示。

【范例解析】 上述代码中，定义了一个整型数组指针 `pa`，其指向数组 `a`，该程序输出了该数组 `a` 的第一个和第二个元素。同时，还定义了一个指针数组 `pb`，其中包含 5 个字符串，该程序输出了该指针数组 `pb` 的第一个和第二个元素。

读者可以看出，使用数组指针访问数组中的元素用首地址加变量 `i` 来表示。例如，访问数组的第二个元素可用表达式 `*(pa[0]+1)` 来表示，其中 `pa[0]` 为数组首地址。而用指针数组访问数组中的元素则直接用数组名即可。例如，访问数组的第二个元素用表达式 `pb[1]` 即可。

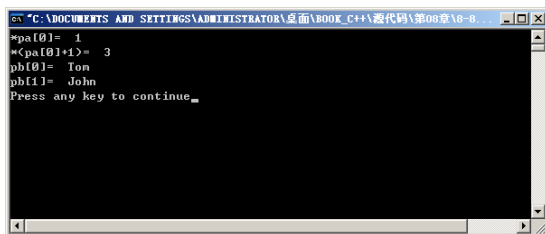


图 8-10 数组指针和指针数组

8.4 指针与函数

在前面的章节中，向读者介绍了在调用函数的实参与形参之间的几种参数传递方式：传值方式、地址传递和引用传递。利用指针做函数参数，可以方便地实现地址传递。函数可以返回指针，指针也可以指向函数。

8.4.1 指针作为函数参数

如果函数的某个参数是指针，对这个函数的调用就是传址调用，也就是使实参指针和形参指针变量指向同一内存地址。在被调用函数的运行过程中，对形参指针所指向的地址中内容的改变也会影响实参。

虽然用指针做函数的参数可以使得形参的改变对相应的实参有效,但是,如果在函数中反复使用指针进行间接访问,不仅会影响程序的可读性,还容易产生错误。因此在 C++ 语言中扩充了引用的概念,这样既可以实现指针所带来的功能,又使程序清晰易读。



提示 当以数据的地址作为实参调用一个函数时,被调用函数的形参也必须是可接收地址的变量,并且数据类型必须与被传送的数据类型相同。由于数组名是一个指针,因此,数组名也可以用做函数的参数,也属于传址调用。

如果在函数定义时将形参的类型说明成指针,对这样的函数进行调用时就需要指定地址值形式的实参。这时的参数传递方式即为地址传递方式。这种地址传递与上述的按值传递不同,它把实参的存储地址传送给对应的形参,从而使得形参指针和实参指针指向同一个地址。因此,被调用函数对形参指针所指向的地址中内容的任何改变都会影响到实参。

【范例 8-9】 指针作为函数的参数。该范例将指针作为函数的参数进行传递,完成两个数之间的互相交换功能,其使用的是地址传递的方式,其实现代码如代码清单 8-9 所示。

代码清单 8-9

```

1  #include <iostream.h>
2  void swap(int *,int *);           //声明参数为指针的函数
3  void main()
4  {
5      int a=3,b=4;                 //定义整型变量并初始化
6      cout<<"a="<<a<<" ,b="
7          <<b<<endl;               //输出变量初始值
8      swap(&a,&b);                  //调用函数
9      cout<<"a="<<a<<" ,b="
10         <<b<<endl;               //输出变量调整后的值
11 }
12 void swap(int *x,int *y)         //定义交换两个变量值的函数 swap
13 {
14     int t=*x;                     //定义指针并交换
15     *x=*y;
16     *y=t;
17 }
```

【运行结果】 在 Visual C++ 中执行上述程序,其结果如图 8-11 所示。

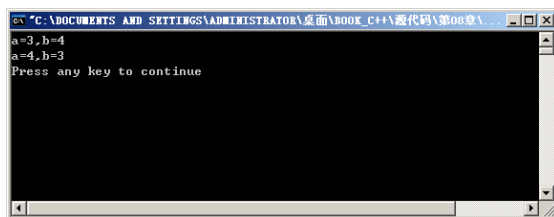


图 8-11 指针作为函数参数

【范例解析】 上述程序中,形式参数是两个整型指针,而在主程序中的实际参数为两个地址: swap(&a,&b),表示变量 a 和 b 的地址。



注意 在函数的传址调用中,传递的参数值并不改变,即指针本身的值并不改变,改变的是它指向的值。



8.4.2 指针型函数

除了 `void` 类型的函数之外，函数在调用结束后都会有返回值，指针同样也可以作为函数的返回值。当一个函数的返回值是指针类型时，这个函数就是指针型函数。

通常非指针型函数调用结束后，可以返回一个变量，但是每次调用只能返回一个数据。但有时需要从被调函数返回一批数据到主调函数中，这时可以通过指针型函数来解决。指针型函数在调用后返回一个指针，通过指针中存储的地址值主调函数就能访问该地址中存放的数据，并通过指针算术运算访问这个地址的前、后内存中的值。因此，通过对空间的有效组织（如数组、字符串等能前后顺序存放多个变量的数据类型），就可以返回大量的数据。定义指针型函数的函数头的一般语法格式为：

数据类型 *函数名（参数表）

其中，参数说明如下：

- 数据类型是函数返回的指针所指向数据的类型。
- *函数名声明了一个指针型的函数。
- 参数表是函数的形参列表。

下面语句声明了一个指针型函数：

```
int *fun(int a,int b);
```

读者可以看出，上述语句表示函数 `fun` 返回一个指针值，这个指针指向一个整型数据，而整型变量 `a`、`b` 是该函数的形参。

【范例 8-10】指针型函数的应用。该范例定义和调用了一个比较两个字符串大小的指针型函数 `max`，该函数的返回值为一个字符型的指针，实现代码如代码清单 8-10 所示。

代码清单 8-10

```
1  #include <iostream.h>
2  #include <string.h>                                //包含头文件 string.h
3  char *max(char *a,char *b)                          //定义指针型函数 max
4  {
5      return(strcmp(a,b)?a:b);                        //返回两个字符串的较大值
6  }
7  void main()
8  {
9      char *p;                                        //定义指针
10     p=max("hello", "good");                         //调用参数 max
11     cout<<p<<endl;
12 }
```

【运行结果】在 Visual C++ 中执行上述程序，其结果如图 8-12 所示。

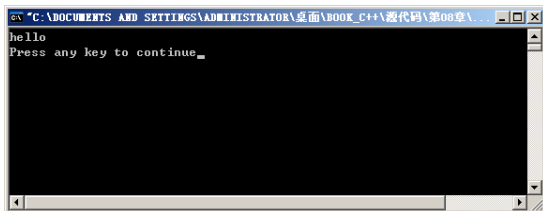


图 8-12 指针型函数

【范例解析】上述程序中，函数 `max` 的返回值是一个指向字符的指针，在主函数 `main()` 中也定义了一个字符型指针用于接收其返回值，并将其输出。



警告 上述程序中使用了字符串比较函数 `strcmp()`，因此在预编译的头文件中需要添加如下语句，否则编译系统将给出错误的提示信息。

```
#include <string.h>
```

8.4.3 函数指针

函数指针就是指向函数的指针。定义函数指针的语法格式为：

数据类型 (*函数指针名)(参数表);

其中，参数定义如下：

- 数据类型是指函数指针所指向函数的返回值的类型。
- 参数表中指明该函数指针所指向函数的形参类型和个数。

例如，下列语句定义了一个函数指针 `p`，其指向一个返回整型值，有两个整型参数的函数。

```
int (*p)(int,int);
```

在定义了指向函数的指针变量后，在使用此函数指针之前，必须先给它赋值，使它指向一个函数的入口地址。由于函数名是函数在内存中的首地址，因此可以将函数名赋给函数指针变量，赋值的一般语法格式为：

函数指针名=函数名;

例如，对上面刚定义的函数指针 `p`，可以给它赋值如下：

```
p=func1;
```

其中，函数名所代表的函数必须是一个已经定义过的，和函数指针具有相同返回类型的函数。并且等号后面只需写函数名而不要写参数。例如，不要写成下列形式：

```
p=func1(a, b);
```

当函数指针指向某函数以后，可以用下列形式调用函数：

(*指针变量)(实参表列)

例如，表达式 `(*p)(a,b)` 就相当于 `func1(a,b)`。



注意 指针的运算在这里是无意义的，因为指针指向函数的首地址。当用指针调用函数时，程序是从指针所指向的位置开始按程序执行，若进行指针运算，程序的执行就不是从函数的开始位置执行，这就会造成错误。

与定义一般变量指针数组一样，C++语言中也可以定义具有特定返回类型和特定参数类型的函数指针数组。函数指针数组也可以是多维的，不过在实际编程中一般只用到一维函数指针数组。定义它的语法格式如下：

数据类型 (*函数指针名[常量表达式])(参数表);

例如，下面语句定义了一个函数指针数组：

```
int (*p[5])(int,int);
```

上述语句定义了一个含有 5 个元素的函数指针数组，其中的每个元素都是一个指向函数的指针，且指向的函数都是返回值类型为整型、带两个整型参数的函数。

【范例 8-11】 函数指针的应用。该范例定义了一个函数指针 `p`，其指向 `max` 函数，在主函数 `main()` 中用该指针调用函数 `max`，其实现代码如代码清单 8-11 所示。



代码清单 8-11

```

1  #include <iostream.h>
2  int max(int x,int y)                                //定义函数 max
3  {
4      return(x>y?x:y);                                //返回两个整数的较大者
5  }
6  void main()
7  {
8      int (*p)(int,int);                            //定义函数指针
9      int a,b,c;                                       //定义变量
10     p=max;                                           //初始化函数指针
11     cout<<"Please input 2 number: " <<endl;
12     cin>>a>>b;                                       //接收用户输入
13     c=(*p)(a,b);                                     //通过函数指针调用函数
14     out<<"max(a,b)= " <<c<<endl;                   //输出
15 }

```

【运行结果】在 Visual C++中执行上述程序，其结果如图 8-13 所示。

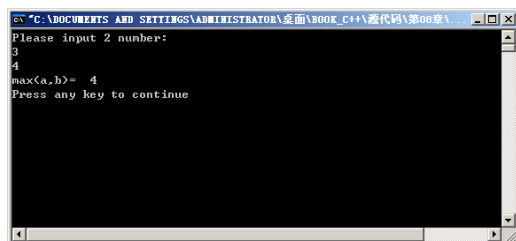


图 8-13 函数指针

【范例解析】上述程序定义了函数 max，其返回两个整型数值的较大者，在 main()函数中定义了函数指针 p，其指向函数 max，并通过该指针调用 max 函数，输出两个整型数值之中的较大者。



警告 函数 max 的参数为两个整型参数，而在 main()函数中定义函数指针 p 时，其也必须包含两个整型参数，即定义格式为 int (*p)(int,int)，否则将导致程序编译出错。

8.5 指针与字符串

第 7 章中提到过，C++中可以定义一个字符数组，实现字符串的存储，并通过数组下标来访问指定字符。事实上，C++还提供了另外一种存储访问字符串的方法，即字符指针，定义一个字符指针，便可以通过该指针的指向来访问指定字符。

如果通过指针访问一个字符串，可以将这个指针指向此字符串，并利用指针的加 1、减 1 操作实现对各个字符的访问。此外，C++提供了许多字符串处理的库函数，在第 7 章介绍数组时也提到过了，其中常用的有：

- strcat(): 字符串连接函数，其原型为 char *strcat(char *s1,char *s2)，将字符串 s2 连接到字符串 s1 的后面，并返回 s1 的地址值。
- strcmp(): 字符串比较函数，其原型为 int strcmp(const char *s1,const char *s2,[int n])，比较字符串 s1 和 s2 的大小（如果有参数 n，比较前 n 个字符的大小）。当字符串 s1 大于、等于或小于字符串 s2 时，函数返回值分别是正数、零和负数。
- strcpy(): 字符串复制函数，其原型为 char *strcpy(char *s1,const char *s2)，将 s2 所指向的字符串复制到 s1 所指向的字符数组中，然后返回 s1 的地址值。

- `strlen()`: 字符串长度计算函数, 原型为 `int strlen(const char *s)`, 该函数返回字符串 `s` 的长度。

【范例 8-12】字符串指针的应用。该范例用字符指针访问字符串, 并使用如上的函数对字符串进行字符串的连接、比较、复制和计算长度操作, 其实现代码如代码清单 8-12 所示。

代码清单 8-12

```

1  #include <iostream.h>
2  #include <string.h>
3  void main()
4  {
5      char str1[100]="hello";           //定义字符串
6      char *str2=" C++";               //定义字符串指针并初始化
7      char *str;                         //定义字符串指针
8      int i=0;
9      cout<<"str2= " <<str2<<endl;
10     str=strcat(str1,str2);              //调用字符串连接函数 strcat
11     cout<<"strcat(str1,str2)= " <<str<<endl;
12     i=strcmp(str1,str2);                 //调用字符串比较函数 strcmp
13     cout<<"strcmp(str1,str2)= " <<i<<endl;
14     str=strcpy(str1,str2);               //调用字符串复制函数 strcpy
15     cout<<"strcpy(str1,str2)= " <<str<<endl;
16     i=strlen(str1);                      //调用字符串长度函数 strlen
17     cout<<"strlen(str1)= " <<i<<endl;
18 }
```

【运行结果】在 Visual C++ 中执行上述程序, 其运行结果如图 8-14 所示。

【范例解析】上述程序首先输出字符串指针 `str2` 指向的值, 依次将字符串 `str1` 和 `str2` 进行连接、比较和复制, 最后计算字符串 `str1` 的长度输出。

需要读者注意的是, 上述程序中对 `str1` 进行声明时没有将其声明成指针, 而是声明成长度为 100 的字符数组, 这是因为 `strcat()`、`strcmp()` 等函数将最后的结果放在 `str1` 中返回, 如果 `str1` 声明为指针, 那么其中的值和长度就已经固定, 当执行字符串连接和复制等操作时若 `str1` 中定义的长度不够, 就会出现运行错误, 如图 8-15 所示。

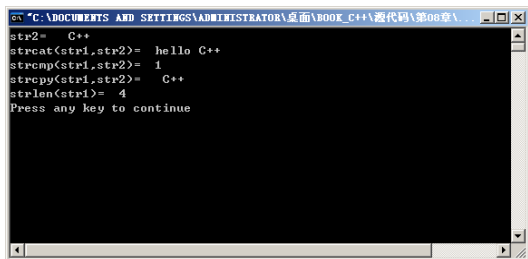


图 8-14 指针与字符串

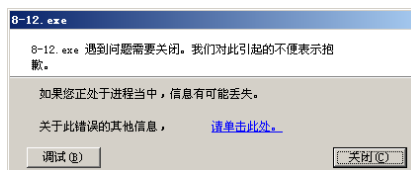


图 8-15 运行时错误



注意 当执行这些字符串函数时, 需定义目标字符串为数组, 而不是指针。此外, 当程序中使用这些字符串处理函数时, 需要在程序的开始加上头文件 `string.h`。

8.6 二级指针

由于指针是一个变量, 在内存中占据一定的空间, 并且具有一个地址, 这个地址也可以利用指针来保存。因此, 可以声明一个指针来指向它, 这个指针称为指向指针的指针, 也称为二

级指针。一般来说，声明指向指针的指针的形式如下：

存储类型 数据类型 **指针变量名

其中，参数说明如下：

- 两个星号“**”表示二级指针。
- 数据类型是指通过两次间接寻址后所访问的变量类型。

例如，下面语句声明了一个指向指针的指针 pp，其指向指针 p。

```
int i,*p=&i;
int **pp=&p;
```

【范例 8-13】 指向指针的指针的应用。该范例定义了一个指向指针的指针 pp，来看一下其输出，其实现代码如代码清单 8-13 所示。

代码清单 8-13

```
1  #include<iostream.h>
2  void main()
3  {
4  int a;
5  int *p=&a,**pp=&p;           //定义指针和指向指针的指针
6  a=1;                         //变量赋值
7  cout<<"a= " <<a<<endl;      //输出变量值
8  out<<"*p= " <<*p<<endl;      //输出指针指向的值
9  cout<<"p= " <<p<<endl;       //输出指针的值
10 cout<<"*pp= " <<*pp<<endl;   //输出指向指针的指针的值
11 cout<<"**pp= " <<**pp<<endl; //输出指向指针的指针指向的值
12 }
```

【运行结果】 上述程序中，定义了指针 p 和指向指针的指针 pp，其中 p 指向整型变量 a，pp 指向 p，该程序其输出了 p 和 pp 的各种值，如图 8-16 所示。

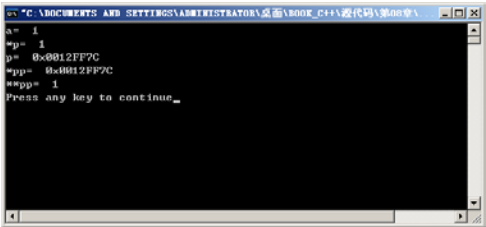


图 8-16 指向指针的指针

【范例解析】 读者可以看出，上述程序中 p 指向的是变量 a，因此*p 的值为 1，p 的值为变量 a 的存储地址。而 pp 指向指针 p，因此*pp 的地址为指针 p 的值，即变量 a 的存储地址，而 **pp 才是变量 a 的值，也即 1。这就符合了上述输出结果。

**提**

在实际的程序中，初学者容易混淆指向指针的指针及指针的值，除非特殊情况，否则一般不建议初学者使用指向指针的指针。

8.7 小结

本章主要介绍了 C++ 中较为复杂的指针的基本内容。以前接触过 C 语言的读者应该知道，指针是 C 语言中最难掌握的，也是最灵活的。本章一开始就通过一个示例介绍了指针的概念和作用，接下来主要介绍了指针的运算，包括通过指针取值（*）、取地址（&）、指针的算术运算

和关系运算等。此外,本章重点介绍了指针的应用,主要包括指针在数组中的应用、在函数中的应用、在字符串中的应用和指向指针的应用。最后,就动态内存分配和引用作了简要介绍。

8.8 习题

1. 写出下列程序的运行结果。

```
#include <iostream.h>
void main()
{
    int *p;
    int n = 100;
    p=&n;
    cout<<"n="<<n<<endl;
    cout<<"*p="<<*p<<endl;
}
```

【解答】该习题主要考查指针的概念。上述语句中定义了指针 p 和整型变量 n , 并为指针 p 进行初始化, 其值为变量 n 的地址。该习题使用间接取值符号 $*$ 将指针指向的值输出, 根据运算符 $*$ 的运算规则, 此处的输出为 100 100。

2. 编写一个 C++ 程序, 接收从键盘输入的 10 个整数, 用指针求这 10 个数中的最大数、最小数和平均值。例如, 输入 45 87 95 31 25 85 94 100 78 37 等 10 个整数, 其输出如图 8-17 所示。

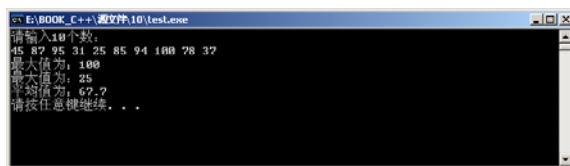


图 8-17 求最大值、最小值和平均值

【解答】该习题主要考查指针与数组的关系。首先定义一个包含 10 个元素的数组和一个指针, 将指针指向该数组, 即将指针初始化为数组的首地址, 数组元素的输入和输出都可以通过指针来实现, 通过循环语句实现所有数组元素的比较。其简要的实现代码如下所示。

```
for(i=0;i<10;i++)
    cin>>*(p+i);
p=a;
p_max=a;
p_min=a;
s=*p;
for(i=1;i<10;i++)
{
    s+=*(p+i);
    if(*(p+i)>*p_max)
        p_max=p+i;
    if(*(p+i)<*p_min)
        p_min=p+i;
}
avg=(float)s/10;
```

3. 编写一个 C++ 程序, 用指针将一个 3*3 的二维矩阵进行转置操作。所谓数组的转置, 即将矩阵的元素根据主次对角线进行交换。例如, 下面是原矩阵和转置后的矩阵。

1	2	3	→	1	4	7
4	5	6		2	5	8
7	8	9		3	6	9



【解答】该习题主要考查指针数组指针的应用。该习题首先需要定义一个二维矩阵和一个数组指针，该指针指向二维数组的首地址，通过双重循环语句输入该数组的元素，通过指针实现数组元素的交换并将交换后的结果输出。其简要的实现代码如下所示。

```
for(i=0;i<3;i++)
    for(j=0;j<3;j++)
        cin>>p[i][j];
for(i=0;i<3;i++)
    for(j=0;j<i;j++)
    {
        t=p[i][j];
        p[i][j]=p[j][i];
        p[j][i]=t;
    }
```

4. 以下程序的输出结果是什么？

```
#include <iostream>
void main()
{
    char s[]="ABCD";
    char *p;
    for (p=s;p<s+4;p++)
        cout<<p<<endl;
}
```

【解答】该习题主要考查字符串与指针的关系。上述语句定义指向字符串的指针 p ，其指向字符串 s 。该循环语句首先将输入字符串 s 的所有字符 $ABCD$ ，因为指针 p 的初值为字符串 s 的首地址，然后将输出字符串 s 中从第二个字符开始的剩余字符串 BCD ，因为指针 p 经过了++运算，其值指向字符串 s 中的字符 B 。依此类推，可以得到该程序的输出结果如下：

```
ABCD
BCD
CD
D
```

5. 编写两个函数，利用函数型指针变量的调用，分别求出两数中的较大值和较小值。

【解答】该习题主要考查指针函数的定义和使用。首先声明两个指针函数分别用于计算两个整数的较大值和较小值，在主函数中调用这两个函数。其简要的实现代码如下所示。

```
int (*p)( ),max( ),min( );
p=max;
c=(*p)(a,b);
p=min;
c=(*p)(a,b);
int max (int x,int y)
{
    int z;
    z=(x>y)?x:y;
    return (z);
}
int min(int x,int y)
{
    int z;
    z=(x<y)?x:y;
    return (z);
}
```

6. 编写一个函数，使用引用作为函数的参数及函数的调用，将 3 个存储在变量中的数值进行降序排列。

【解答】该程序可定义包含两个引用作为参数的函数 fun ，其用于交换两个变量的值，再定

义包含三个引用作为参数的函数 `exchange`，在该函数中又调用了 `fun` 函数，实现三个变量的值的交换。执行该函数后，将三个变量中最大的值存储在变量 `a` 中，次大的值存储在变量 `b` 中，而最小的值存储在变量 `c` 中。其简要的实现代码如下所示。

```
void fun(int &a,int &b)                //定义参数为引用的函数
{
    int p;                            //定义整型变量
    p=a;                              //交换两个数值
    a=b;
    b=p;                              //交换完成
}
void exchange(int &a,int &b,int &c)    //定义包含三个引用参数的函数
{
    if(a<b)                           //a<b 成立
        fun(a,b);                    //交换 a, b
    if(a<c)                           // a<c 成立
        fun(a,c);                    //交换 a, c
    if(b<c)                           //b<c 成立
        fun(b,c);                    //交换 b, c
}
void main()
{
    int a,b,c;                        //定义整型变量
    a=12;                             //整型变量初始化
    b=639;
    c=78;
    exchange(a,b,c);                  //调用函数 exchange
    cout<<"a="<<a<<" ,b="<<b<<" ,c="<<c<<endl; //输出变量的值
}
```

第 9 章 构造数据类型

构造数据类型是根据已定义的一个或多个数据类型用构造的方法来定义的。也就是说，一个构造类型的值可以分解成若干个“成员”或“元素”。每个“成员”都是一个基本数据类型或是一个构造类型。在 C++ 中，构造类型有以下几种：

- 数组类型
- 结构体类型
- 共用体类型
- 枚举类型
- 用户自定义数据类型

关于数组类型，在第 7 章已经详细介绍过，此处不再赘述。本章将重点介绍 C++ 中的结构体、共用体、枚举和用户自定义等几种数据类型。以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为一个学习的参照。

- 掌握 C++ 中常见的几种构造数据类型：结构体、共用体和枚举类型的定义和使用。
- 了解类型重定义符的使用。
- 了解位域的应用。

9.1 结构体

在 C++ 中，结构体是一种可以由程序员根据实际情况来自己构造的新的数据类型，结构体类型的数据由若干称为“成员”的数据组成，每一个成员既可以是一个基本数据类型的数据，也可以是另一个构造类型的数据。

9.1.1 结构体概述

无论是数组还是基本数据类型都仅仅描述了事物某一方面的特性，但是，一种物体或事物往往具有多方面的属性。例如，描述一个同学可能要包括学号、姓名、性别、年龄、成绩、班级等多方面的信息。根据前面的知识，需要定义一组变量来描述上述信息。

```
int Code;           //学号
char Name[20];      //姓名
char Sex;           //性别
int Age;            //年龄
...
```

上述几个变量之间在组织和存储上没有强制的约束关系，不能够将其作为一个整体来对待，而这些属性在逻辑上属于同一事物，应当作为一个整体来处理。在 C++ 中引入了一种新的自定义数据类型——结构体（structure）。引入结构体之后，程序设计人员可以根据需要定义多种自定义的数据类型，用于描述不同类型的事物。

例如，针对上面描述一个同学所需的各种信息，可以定义一个结构体，其包括学号、姓名、性别、年龄、成绩、班级等信息，如图 9-1 所示。

简单地说，结构体就是一个可以包含不同数据类型的结构，其是一种可以自己定义的数据类型，它和数组主要有两点不同：

- 结构体可以在一个结构中声明不同的数据类型。
- 相同结构的结构体变量是可以相互赋值的，而数组是做不到的。

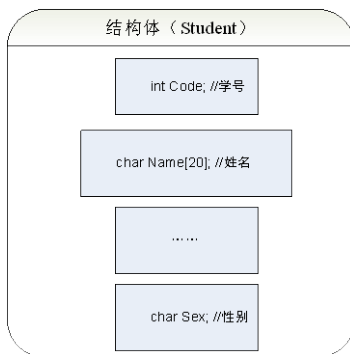


图 9-1 结构体



注意 因为数组是单一数据类型的数据集合，它本身不是数据类型（而结构体是），数组名称是常量指针，所以不可以作为左值进行运算，所以数组之间就不能通过数组名称相互赋值了，即使数组数据类型和数组大小完全相同。

9.1.2 结构体类型说明

掌握了结构体的概念后，在实际程序中就可以定义结构体这一数据类型了。作为一种自定义的数据类型，在使用结构体之前，必须完成其定义。与定义基本数据类型一样，定义结构体也需要一个关键字，定义结构体使用的是 **struct** 关键字。一般来说，说明结构体类型的语法格式如下：

```
struct 结构体标识符
{
    成员变量列表;
    ...
};
```

其中，参数说明如下。

- **struct 关键字**：它为系统关键字，用于说明当前定义一个新的结构体类型。
- **结构体标识符**：为遵循 C++ 标识符命名规则的一个标识符。
- **成员变量列表**：在 {} 之间通过分号分割的变量列表称为成员变量（**structure member**），用于描述此类事物的某一方面特性。成员变量可以为基本数据类型（如 **float**）、数组和指针类型，也可以为结构体。由于不同的成员变量分别描述事物某一方面的特性，因此成员变量不能重名。

例如，为了 9.1.1 节中包含学号、姓名、性别、年龄等信息的学生，可以定义如下的自定义数据类型 **struct Student**。

```
struct Student
{
    int Code;           //学号
    char Name[20];      //姓名
    char Sex;           //性别
    int Age;            //年龄
};
```

上述定义的结构体 **Student** 中的成员变量都是基本数据类型，如 **int**、**char** 等。事实上，在



结构体的定义中，不但可以包含基本数据类型，还可以包含结构体数据类型。例如，为了描述三维世界中的坐标点，可以定义结构体 `struct Point` 如下：

```
struct Point
{
    double x;           //x 坐标
    double y;           //y 坐标
    double z;           //z 坐标
};
```

而为了描述三维世界中的直线信息，可以定义下面的结构体 `struct Line`：

```
struct Line
{
    struct Point StartPoint; //起始点
    struct Point EndPoint;   //结束点
};
```



提示 结构体 `struct Line` 中的成员变量就不像 `int` 和 `char` 等那样为基本数据类型了，而是 `struct` 结构体类型，这在实际程序中也是应用较多的。

9.1.3 定义结构体类型变量

在完成一个结构体定义之后，就可以像定义基本数据类型变量一样定义结构体类型的变量和数组了。一般来说，结构体类型变量的定义可以通过如下 4 种方式完成。

1. 先定义结构体类型再单独进行变量定义

例如，说明一个结构体类型 `Student`，并定义一个该结构体类型的变量 `Stu`、一个该结构体类型的数组 `Stu[10]` 和一个该结构体类型的指针 `*pStu`，如下所示：

```
struct Student
{
    int Code;           //学号
    char Name[20];      //姓名
    char Sex;           //性别
    int Age;            //年龄
};
struct Student Stu      //定义 struct Student 类型变量
struct Student Stu [10]; //定义类型数组
struct Student *pStu;    //定义类型指针
```

读者注意到了，此处先说明了结构体类型 `struct Student`，再由一条单独的语句定义了结构体变量 `Stu`、结构体数组 `Stu[10]` 和结构体指针 `*pStu`，该语句的“`struct Student`”代表类型名，如同用 `int` 定义变量时和 `int` 是类型名一样，该语句的 `struct Student` 相当于 `int` 的作用，因此 `struct` 必须与结构标识名共同来说明不同的结构体类型，`struct` 和 `Student` 都不能省略。定义了一个结构体类型后，可以多次用它来定义变量。

上述语句声明结构体 `struct Student` 的变量 `Stu` 后，该变量的存储结构如图 9-2 所示。

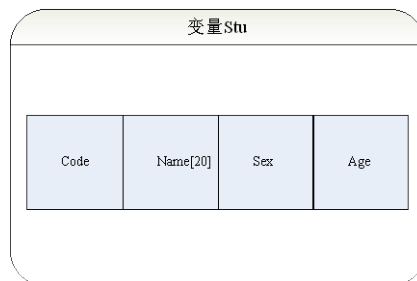


图 9-2 结构体存储结构

2. 紧跟在结构体类型说明之后进行定义

同样,下面语句说明一个结构体类型 `Student`,并定义了一个该结构体类型的变量 `Stu` 和一个该结构体类型的数组 `Stu[10]`及一个该结构体类型的指针 `*pStu`。

```
struct Student
{
    int Code;                //学号
    char Name[20];           //姓名
    char Sex;                //性别
    int Age;                 //年龄
}Stu,Stu[10],* pStu;
```

这里在说明结构体类型 `struct Student` 的同时,定义了一个结构体变量 `Stu` 和具有 10 个元素的结构体数组 `Stu[10]`。此外,还可以用 `struct Student` 再定义其他变量,例如:

```
struct Student Stu1,Stu2;
```

3. 在说明一个无名结构体类型的同时直接进行定义

同样,下面语句说明一个结构体类型 `Student`,并定义了一个该结构体类型的变量 `Stu` 和一个该结构体类型的数组 `Stu[10]`及一个该结构体类型的指针 `*pStu`。

```
struct
{
    int Code;                //学号
    char Name[20];           //姓名
    char Sex;                //性别
    int Age;                 //年龄
}Stu,Stu[10], *pStu;
```



注意 此时只是直接定义了 `Stu, Stu[10]`为上面花括号内的结构体类型,但没有定义此结构体类型的名字,因此不能再用它来定义其他变量。例如,下面的定义是不合法的。

```
struct Stu1,Stu2;
```

4. 使用 typedef 说明一个结构体类型名后再用新类型名来定义变量

下面语句说明一个结构体类型 `Student`,并定义了一个该结构体类型的变量 `Stu` 和一个该结构体类型的数组 `Stu[10]`及一个该结构体类型的指针 `*pStu`。

```
typedef struct
{
    int Code;                //学号
    char Name[20];           //姓名
    char Sex;                //性别
    int Age;                 //年龄
}Student;
Student Stu,Stu[10], *pStu;
```

这里的 `Student` 是一个具体的结构体类型名,其能够唯一地标识这种结构体类型。因此,可用它来定义变量,如同使用 `int`、`char` 一样,不可再写关键字 `struct`。关于 `typedef` 说明用户自定义数据类型,在 9.4 节中还将详细讲解。

9.1.4 初始化结构体变量

C++中引用变量的基本原则是在使用变量前,需要对变量进行定义并初始化,方法是在定义变量的同时给其赋一个初始值。结构体变量的初始化,遵循相同的规律。结构体变量的初始化方式与数组类似,分别给结构体的成员变量以初始值,而结构体成员变量的初始化遵循简单



变量或数组的初始化方法。具体的形式如下：

```
struct 结构体标识符
{
    成员变量列表;
    ...
};
struct 结构体标识符 变量名={初始化值 1,初始化值 2,..., 初始化值 n };
```

例如，下列语句对上述说明的结构体 struct Student 定义的变量 Stu 进行初始化：

```
struct Student
{
    int Code;                //学号
    char Name[20];           //姓名
    char Sex;                //性别
    int Age;                 //年龄
}
struct Student Stu={200301,"张三",'M',21};
```

上述语句完成对变量 Stu 初始化后，该变量的存储结构如图 9-3 所示。




图 9-3 初始化变量存储结构

由于定义结构体变量有多种方法，因此初始化结构体变量的方法对应也有多种，上面已经介绍了其中的一种形式，下面再介绍两种方法：

```
struct Student
{
    int Code;                //学号
    char Name[20];           //姓名
    char Sex;                //性别
    int Age;                 //年龄
}Stu={200301,"张三","男",21}

struct
{
    int Code;                //学号
    char Name[20];           //姓名
    char Sex;                //性别
    int Age;                 //年龄
}Stu={200301,"张三",'M',21}
```



注意 对结构体变量进行赋初值时，C++编译程序按每个成员在结构体中的顺序一一对应赋初值，不允许跳过前面的成员给后面的成员赋初值。但是，C++允许只给前面的若干个成员赋初值，对于后面未赋初值的成员，系统自动赋默认的初值。例如：



```
struct Student
{
    int Code;                //学号
    char Name[20];           //姓名
    char Sex;                //性别
    int Age;                 //年龄
}Stu={200301};
```

其相当于给 code 变量赋值 200301, 而其他的 Name 变量赋值为空, Sex 变量赋值 '\0x0', Age 变量赋值为 0。如果仅仅对其中部分的成员变量进行初始化, 要求初始化的数据至少有一个, 其他没有初始化的成员变量由系统完成初始化, 为其提供默认的初始化值。各种基本数据类型的成员变量初始化默认值如表 9-1 所示。

表 9-1 基本数据类型成员变量的初始化默认值

数据类型	默认初始化值
int	0
char	'\0x0'
float	0.0
double	0.0
char Array[n]	""
int Array[n]	{0,0,...,0}

此外, 对于复杂结构体类型变量的初始化, 同样遵循上述规律, 对结构体成员变量分别赋予初始化值。例如:

```
struct Line
{
    int id;
    struct Point StartPoint;
    struct Point EndPoint;
}oLine1={0,                //初始化 id
        {0,0,0},           //初始化 StartPoint
        {100,0,0}}         //初始化 EndPoint
};
```

其中常量 0 用于初始化 oLine1 的基本类型成员变量 id; 常量列表 {0,0,0} 用于初始化 oLine1 的 struct Point 类型成员变量 StartPoint; 常量列表 {100,0,0} 用于初始化 oLine1 的 struct Point 类型成员变量 EndPoint。

9.1.5 引用结构体成员变量

前面介绍过一个结构体变量中包括一个或多个成员变量, 在实际使用中, 就需要对其成员变量进行引用。一般来说, 如果已定义了一个结构体变量和一个指向该结构体的指针变量, 则可用以下三种形式来引用结构体变量中的成员:

- 结构体变量名 . 成员名
- 指针变量名->成员名
- (*指针变量名). 成员名

其中, 结构体变量名也可以是已定义的结构体数组的数组元素, 指针变量名为定义的同一种结构类型的指针变量, 并使该指针指向同类型的变量。



提示 上述格式中的 “.” 称为成员运算符, “->” 称为结构指向运算符。



【范例 9-1】引用结构体成员变量。该范例说明了一个结构体 `Student`，定义了变量 `Stu` 和指向结构体 `Student` 的指针 `pStu`，在主程序中引用该结构体的成员变量，实现代码如代码清单 9-1 所示。

代码清单 9-1

```
1  #include <iostream.h>
2  struct Student
3  {
4      int Code;                //学号
5      char Name[20];           //姓名
6      char Sex;                //性别
7      int Age;                 //年龄
8  };
9  struct Student Stu={200301,"张三",'M',21};    //定义结构体变量
10 void main()
11 {
12     struct Student *pStu=&Stu;                //定义结构体指针
13     cout<<"Stu.Code= "<<Stu.Code<<endl;        //引用结构体成员变量的第一种形式
14     cout<<"pStu->Code= "<<pStu->Code<<endl;    //引用结构体成员变量的第二种形式
15     cout<<"(*pStu).Code= "<<(*pStu).Code<<endl; //引用结构体成员变量的第三种形式
16 }
```

【运行结果】在 Visual C++ 中运行上述程序，其结果如图 9-4 所示。

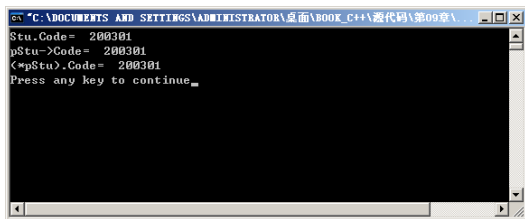


图 9-4 引用结构体成员变量

【范例解析】上述代码中，首先定义了一个结构体 `Student`，其包含 4 个成员变量，再定义了一个该结构体的变量 `Stu`，然后分别用了如上述的三种引用成员变量方式引用其中的变量，其得到的结果都是相同的。



注意 如果定义的结构体变量是数组类型，那么引用就稍有不同。

【范例 9-2】引用结构体数组变量的成员变量。该范例将上述代码 9-1 稍作修改，实现输出数组变量中的值，代码如代码清单 9-2 所示。

代码清单 9-2

```
1  #include <iostream.h>
2  struct Student
3  {
4      int Code;                //学号
5      char Name[20];           //姓名
6      char Sex;                //性别
7      int Age;                 //年龄
8  };
9  struct Student Stu[2]={200301,"张三",'M',21},{200302,"李四",'M',23}}; //定义结构体变量并初始化
10 void main()
```

```

11 {
12     struct Student *pStu;                //定义结构体指针
13     for(pStu=Stu;pStu<Stu+2;pStu++)      //循环输出结构体中变量的值
14     {
15         cout<<pStu->Code<<"\t"<<pStu->Name<<"\t"<<pStu->Sex<<"\t"<<pStu->
Age<<endl;
16     }
17 }

```

【运行结果】在 Visual C++中执行上述程序，其结果如图 9-5 所示。

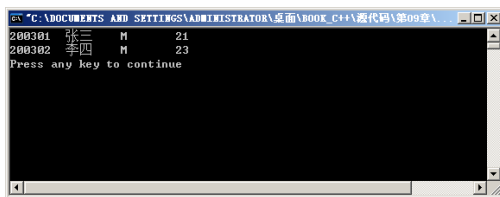


图 9-5 引用结构体数组变量

【范例解析】上述程序中，定义了数组变量 `Stu`，其中有 2 个数组元素，并在定义时候将其初始化，在主程序中调用 `for` 循环语句将其中元素输出，此处采用的是“`->`”符号引用结构体中的成员变量。

事实上，C++中除了可以引用结构体的成员变量外，还可以对结构体变量的本身进行引用，这里就不再赘述，有兴趣的读者可参考相关资料。

9.1.6 结构体作为函数参数

前面介绍数据类型时提到了该类型作为函数参数的传递，如数组等。同样，结构体也可以作为函数的参数进行传递。一般来说，结构体作为函数参数使用的是引用传递。

这是因为利用引用传递的好处很多，其效率和指针相差无几，但引用的操作方式和值传递几乎一样。种种优势都说明善用引用可以使程序易读和易操作，它的优势是当结构体很大的时候，避免传递结构体变量很大的值，节省内存，提高效率。

【范例 9-3】结构体作为函数参数。该范例定义了一个结构体和一个实现输出函数，并将该结构体作为函数参数，在主函数中调用该函数，其实现代码如代码清单 9-3 所示。

代码清单 9-3

```

1  #include <iostream.h>
2  #include <string.h>
3  struct test                                //定义结构体
4  {
5      char name[10];                        //定义姓名
6      float score;                          //定义分数
7  };
8  void print_score(test &pn)                //以结构变量进行传递
9  {
10     cout<<pn.name<<"\t"<<pn.score<<endl;    //输出结构体中分数成员的值
11 }
12 void main()
13 {
14     test a[2]={{"张三",88.5},{ "李四",98.5}}; //定义结构体数组变量并初始化
15     int num = sizeof(a)/sizeof(test);         //计算数组长度
16     for(int i=0;i<num;i++)                   //进入循环
17     {

```



```
18         print_score(a[i]);           //输出分数
19     }
20 }
```

【运行结果】在 Visual C++ 中运行上述程序，其结果如图 9-6 所示。

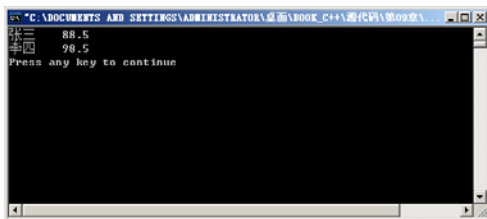


图 9-6 结构体作为函数参数

【范例解析】上述程序中，定义了结构体 `test`，其包含姓名（`name`）和分数（`score`）两个成员变量。此外，又定义了函数 `print_score`，用于输出该结构体中两个变量的值。在主函数 `main()` 中定义该结构体的数组变量 `a`，其中包含 2 个元素，并对其进行了初始化，最后通过 `for` 循环语句输出该数组中两个元素包含的成员变量的值。



提 上述结构体 `test` 在函数 `print_score` 中作为形式参数，并使用了 `&` 符号，表示参数传递时采用引用传递，在主函数 `main()` 中的实参直接用值即可。

9.2 共用体

在具体的程序设计中，有时需要将几种不同类型的变量存放到同一段单元中，或者说，需要使几个不同的变量共占同一段内存。在 C++ 中，提供了这样一种类型结构，即共用体。在有些参考资料上，共用体也称为联合（`Union`）。

9.2.1 共用体类型说明

共用体类型的说明与结构体类型说明方式完全相同。不同的是，结构体变量中的成员各自占有自己的存储空间，而共用体变量中的所有成员占有同一个存储空间。因此，说明一个共用体的语法格式如下所示：

```
union 共用体标识符
{
    成员变量列表;
    .....
};
```

其中，参数说明如下。

- **union 关键字：**其为系统的关键字，作用是通知系统目前定义了一个名为“共用体标识符”的共用体。
- **共用体标识符：**其可以是任何符合 C++ 标识符定义规则的标识符。
- **成员变量列表：**其中的成员列表可以是任何类型的变量。

例如，下面语句定义了一个共用体，其包含两个成员变量。

```
union variant
{
    int ival;
    float fval;
};
```



注意 上述语句定义的共用体 variant 包含两个成员变量，整型数据类型的 ival 变量和浮点型数据类型的 fval 变量，它们共用一个存储空间。

9.2.2 定义共用体类型变量

与结构体类型变量的定义相似，共用体类型变量的定义方法可以采用以下四种方式。

1. 说明共用体时定义

例如，下面语句定义的共用体变量 s1 和 s2 就是在说明共用体类型后直接定义，类型定义没有和变量定义分开。

```
union un
{
    float x;
    int i;
}s1,s2;
```

2. 共用体类型定义与变量分开

例如，下面语句定义了共用体变量 s1 和 s2，这两个变量是在说明共用体类型后再定义的，即类型定义和变量定义分开。

```
union un
{
    float x;
    int i;
}
union un s1, s2;
```

3. 直接定义共用体变量

例如，下面语句定义的共用体变量 s1 和 s2 就是直接定义。

```
union
{
    float x;
    int i;
}s1,s2;
```

4. 采用 typedef 重定义类型

例如，下面语句定义的共用体变量 s1 和 s2 就是采用 typedef 重定义类型定义的。

```
typedef union
{
    float x;
    int i;
} un
un s1,s2;
```



提示 上面这四种方法定义的变量 s1 和 s2 都是相同的。由于共用体使用的是同一个存储空间，因此定义上面的变量后，s1 和 s2 的存储结构如图 9-7 所示。

9.2.3 引用共用体成员变量

与结构体成员变量的引用类似，定义了共用体变量和指向该共用体的指针变量后，共用体变量中每个成员的引用可以使用以下三种形式之一：



- 共用体变量名.成员名
- 指针变量名->成员名
- (*指针变量名).成员名

【范例 9-4】引用共用体成员变量。该范例定义了一个共用体变量和一个指针，给该共用体中的成员变量赋值，最后输出到屏幕，实现代码如代码清单 9-4 所示。

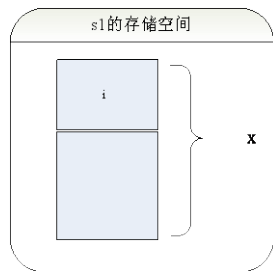


图 9-7 共用体存储结构体

代码清单 9-4

```

1  #include <iostream.h>
2  union variant                                //定义共同体
3  {
4      int ival;                                //定义其成员变量
5      float fval;
6      double dval;
7  };
8  union variant s={5};                        //定义共用体变量并初始化
9  void main()
10 {
11     union variant *p=&s;                    //定义执行共用体变量 s 的指针 p
12     cout<<"s.ival= "<<s.ival<<endl;        //输出共用体中的成员变量的第一种方式
13     cout<<"p->fval= "<<p->fval<<endl;      //输出共用体中的成员变量的第二种方式
14     cout<<"(*p).dval= "<<(*p).dval<<endl; //输出共用体中的成员变量的第三种方式
15 }
```

【运行结果】在 Visual C++中执行上述程序，其结果如图 9-8 所示。

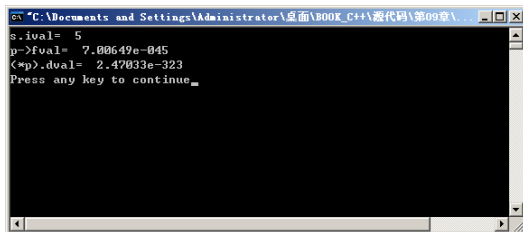


图 9-8 引用共用体成员变量

【范例解析】上述程序中，定义了共用体变量 s，并为其赋初值 5。需要读者注意的是，由于共用体是共用一个存储空间，因此赋初值时只能赋一个值，通常编译系统中认为是第一个成员变量的值。因此，上述语句 union variant s={5}表示 s.ival=5，而其他的成员变量值都是未知的。



注意 读者可以看出，由于共用体共用一个存储空间，因此该共用体中的三个成员变量只有第一个有值，其他两个值均为随机值，是没有意义的。

此外，与结构体变量类似，共用体变量也可以进行整体赋值。C++中，允许在两个类型相同的共用体变量之间进行赋值操作。

【范例 9-5】共用体变量的整体赋值。该范例将代码 9-4 稍作修改，定义两个共用体变量 s1 和 s2，为 s1 赋值 5，执行 s2=s1 的整体赋值语句，代码如代码清单 9-5 所示。

代码清单 9-5

```

1  #include <iostream.h>
2  union variant                                //定义共用体
3  {
4      int ival;                                //定义共用体成员变量
5      float fval;
6      double dval;
7  };
8  union variant s1={5};                        //定义共用体变量并赋值
9  void main()
10 {
11     union variant *p=&s1;                    //定义指向共用体变量的指针
12     union variant s2;
13     s2=s1;                                    //共用体整体赋值
14     cout<<"p->ival= " <<p->ival<<endl;      //输出查看结果
15     cout<<"s2.ival= " <<s2.ival<<endl;
16 }

```

【运行结果】在 Visual C++ 中执行上述程序，其返回结果如图 9-9 所示。

【范例解析】读者可以看出，上述代码中的 p 指向的是变量 s1，通过整体赋值语句"s2=s1;" 给 s2 也赋值了，因此其执行结果如图 9-9 所示。

关于共用体其他的应用，如作为函数中的参数等，基本都与结构体的应用类似，这里就不再赘述了，有兴趣的读者可参考相关资料。

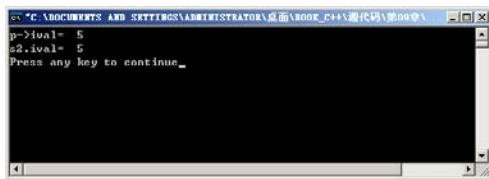


图 9-9 共用体变量的整体赋值

9.3 枚举

在日常生活中，会遇到很多集合类问题，其所描述的状态为有限几个，如比赛的结果只有输和赢两种状态，一周有 7 天，共 7 个状态。在计算机中表述这些信息，需要定义一组整型常量，但是这些常量虽然表达了同一类型的信息，但是在语法上是彼此孤立的个体，而不是一个完整的逻辑整体。因此，C++ 中引入了枚举类型来将这些常量融合成一个整体。

9.3.1 定义枚举类型

一般来说，枚举类型的定义的语法描述如下：

```
enum 枚举标识符 { 常量列表 };
```

其中，参数说明如下。

- **enum 关键字：**为系统关键字，表示定义枚举类型。
- **枚举标识符：**为一个遵循变量的命名规则的标识符。
- **常量列表：**常量列表包含该枚举类型的值，每个枚举常量之间通过逗号分隔。

例如，在实际程序需要描述上、下、前、后、左和右的 6 个方位，那么可定义枚举类型 enum Direction，如下所示：

```
enum Direction {up,down,before,back,left,right};
```



其中, up、down、before、back、left、right 为枚举常量, 可以直接引用。此外, 还可以为枚举常量指定其对应的整型常量数值, 例如:

```
enum Direction {up=1,down=2,before=3,back=4,left=5,right=6};
```



警告 枚举常量值不能出现重复, 如下面的情况是非法的, enum Direction{up=1, down=1,before=3,back=4, left=5,right=6};

如果在给定枚举常量的时候不指定其对应的整数常量值, 系统将自动为每一个枚举常量设定对应的整数常量值。例如, 下列定义的枚举类型中 up 对应的整数值为 0, down 对应的整数值为 1, 以此类推 right=5, 这是 C++ 系统默认的, 由编译系统自动赋值。

```
enum Direction{up, down, before, back, left, right};
```

另外, 允许设定部分枚举常量对应的整数常量值, 但是要求从左到右依次设定枚举常量对应的整数常量值, 并且不能重复。例如:

```
enum Direction{up=7,down=1,before,back,left,right};
```

则从第一没有设定值的常量开始, 其整数常量值为前一枚举常量对应的整数常量的值加 1。

【范例 9-6】 枚举常量对应的整数值。该范例给出了枚举常量对应的整数常量值的应用实例, 实现代码如代码清单 9-6 所示。

代码清单 9-6

```
1  #include <iostream.h>
2  enum Direction1 {up1,down1,before1,back1,left1,right1};    //定义枚举变量
3  enum Direction2 {up2=7,down2=2,before2,back2,left2,right2}; //定义枚举变量
4  void main()
5  {
6      cout<<"Direction1.up1= "<<up1<<endl;                //输出 Direction1
                                                                //中枚举常量对应的整数值
7      cout<<"Direction1.down1= "<<down1<<endl;
8      cout<<"Direction1.before1= "<<before1<<endl;
9      cout<<"Direction1.back1= "<<back1<<endl;
10     cout<<"Direction1.left1= "<<left1<<endl;
11     cout<<"Direction1.right1= "<<right1<<endl;
12     cout<<"Direction2.up2= "<<up2<<endl;                //输出 Direction2 中的枚举常量
                                                                //对应的整数值
13     cout<<"Direction2.down2= "<<down2<<endl;
14     cout<<"Direction2.before2= "<<before2<<endl;
15     cout<<"Direction2.back2= "<<back2<<endl;
16     cout<<"Direction2.left2= "<<left2<<endl;
17     cout<<"Direction2.right2= "<<right2<<endl;
18 }
```

【运行结果】 在 Visual C++ 中执行上述程序, 其运行结果如图 9-10 所示。

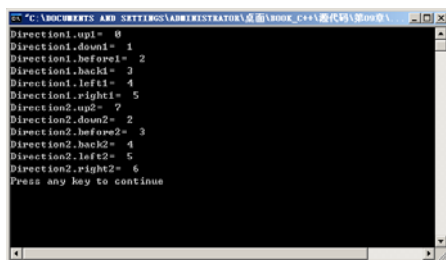


图 9-10 枚举常量对应值

【范例解析】上述程序中，分别定义了两个枚举类型，在主函数中将其值输出。其中，第一个枚举变量 `Direction1` 中没有给枚举常量赋值，第二个枚举变量 `Direction2` 中则给枚举常量的前两个赋了值，后面的常量则没有赋值。



读者可以看出，当采用默认值时，枚举常量的对应值从 0 开始，而部分常量已经赋值了，则其余枚举常量的对应值为前一枚举常量对应的整数常量的值加 1。

9.3.2 定义枚举类型变量

定义了枚举类型后，就可以接着定义枚举类型的变量了。与结构体变量和共用体变量的定义类似，枚举类型的变量定义需要先定义枚举类型，一般有如下两种方式。

- 先定义枚举类型，然后再定义枚举变量。下列语句首先定义了枚举类型 `Direction`，其包含 6 个枚举常量，再定义了 2 个枚举类型变量 `fisrt Direction` 和 `second Direction`。

```
enum Direction{up,down,before,back,left,right};
enum Direction fisrt Direction,second Direction;
```

- 定义枚举类型，同时定义枚举变量。例如，下列语句定义了一个包含 6 个枚举常量的枚举类型 `Direction`，同时，该语句还定义了 2 个枚举类型变量 `fisrt Direction` 和 `second Direction`。

```
enum Direction{up,down,before,back,left,right} fisrt Direction,second Direction ;
```

采用如上两种方式定义的枚举变量是相同的。

【范例 9-7】定义枚举类型变量。该范例分别采用如上的这两种方式定义了 2 个枚举变量，其实现代码如代码清单 9-7 所示。

代码清单 9-7

```
1  #include <iostream.h>
2  void main()
3  {
4      enum Direction1{up1,down1,before1,back1,left1,right1};
5      enum Direction1 first;           //定义枚举变量 first
6      cout<<"fisrt Direction is defined"<<endl;
7      enum Direction2{up2,down2,before2,back2,left2,right2} second;
8      cout<<"second Direction is defined"<<endl;
9  }
```

【运行结果】在 Visual C++ 中执行上述程序，其结果如图 9-11 所示。

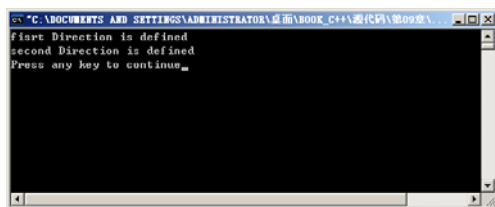


图 9-11 定义枚举类型变量

【范例解析】上述代码中，定义了枚举变量 `first` 和 `second`，其中 `first` 变量的定义采用了先定义枚举类型后定义枚举变量的方式，`second` 变量的定义采用了定义枚举类型同时定义枚举变量的方式，但定义结果是相同的。



9.3.3 引用枚举类型变量

通过 9.3.2 节的学习,读者知道枚举类型的实质是整数集合。因此,枚举变量的引用类似于 int 类型变量的引用,并可以与整数类型的数据之间进行类型转换。

【范例 9-8】引用枚举类型变量。该范例定义描述颜色的枚举类型 enum COLORS, 并进行相关的运算, 其实现代码如代码清单 9-8 所示。

代码清单 9-8

```
1  # include <iostream.h>
2  enum COLORS                                //定义枚举
3  {
4      BLACK,BLUE,GREEN,CYAN,RED,MAGENTA,BROWN,LIGHTGRAY,DARKGRAY,LIGHTBLUE,
LIGHTGREEN,LIGHTCYAN,LIGHTRED,LIGHTMAGENTA,YELLOW,WHITE
5  };                                          //定义枚举类型
6  void main()
7  {
8      enum COLORS a,b,c;                    //定义枚举变量
9      enum COLORS *pa;                     //定义枚举指针
10     int d=-2;
11     a=RED;                                //枚举变量赋值
12     b=WHITE;
13     c=BLACK;
14     cout<<"a= "<<a<<endl;                //输出枚举变量对应的值
15     cout<<"b= "<<b<<endl;
16     cout<<"c= "<<c<<endl;
17     cout<<"d= "<<d<<endl;
18     int e=a+b;                            //枚举常量的运算
19     if(a>b)
20         pa=&a;                             //枚举指针初始化
21     else
22         pa=&b;
23     cout<<"*pa= "<<*pa<<endl;              //输出
24     cout<<"e= "<<e<<endl;
25 }
```

【运行结果】在 Visual C++ 中执行上述代码, 其运行结果如图 9-12 所示。

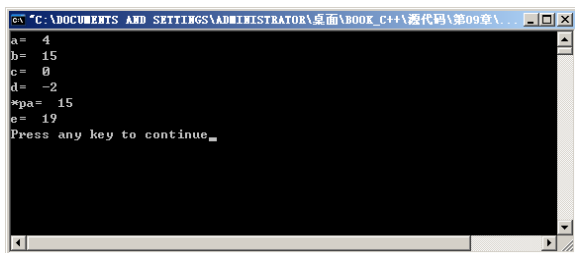


图 9-12 引用枚举类型变量

【范例解析】上述代码中, 定义了三个枚举变量 a、b、c, 分别给其赋值枚举常量, 将其作为整数进行算术运算和关系运算, 并将计算结果输出。

提 读者可以看出, 枚举变量的使用和整数常量的使用一样, 其每个枚举常量对应了一个整数常量, 并且可以进行各种整数常量的运算。

9.4 类型重定义 typedef

在现实生活中, 信息的概念可能是长度、数量和面积等。在 C++ 中, 信息被抽象为 `int`、`float` 和 `double` 等基本数据类型。从基本数据类型名称上, 不能够看出其所代表的物理属性, 并且 `int`、`float` 和 `double` 为系统关键字, 不可以修改。为了解决用户自定义数据类型名称的需求, C++ 中引入类型重定义语句 `typedef`, 可以为数据类型定义新的类型名称, 从而丰富数据类型所包含的属性信息。

在 C++ 中, 类型重定义 `typedef` 的一般语法描述如下:

```
typedef 类型名称 类型标识符;
```

其中, 参数说明如下。

- `typedef`: 为系统保留字, 表示类型重定义。
- 类型名称: 为已知数据类型名称, 包括基本数据类型和用户自定义数据类型。
- 类型标识符: 为新的类型名称。

例如, 下列语句定义了两个新的类型 `LENGTH` 和 `COUNT`, 其分别对应的是基本数据类型中的双精度数据类型 `double` 和短整数 `unsigned int` 类型。

```
typedef double LENGTH;
typedef unsigned int COUNT;
```

经过上述定义新的类型名称之后, 在实际的程序中就可以像用基本数据类型那样定义变量。例如:

```
LENGTH a;
COUNT b;
```

其分别相当于如下语句:

```
double a;
unsigned int b;
```

一般来说, 类型重定义 `typedef` 的主要应用有如下的几种形式。

1. 为基本数据类型定义新的类型名

该形式的应用主要是丰富数据类型中包含的属性信息, 重定义的新类型可表示某种有意义的含义。此外, 在系统移植时, 也需要用到该形式, 例如:

```
typedef unsigned int COUNT;
typedef double AREA;
```

2. 为自定义数据类型(结构体、共用体和枚举类型)定义简洁的类型名称

在前面介绍的结构体、共用体和枚举类型等构造数据类型时, 其定义变量往往较为复杂。例如, 下列语句定义了两个结构体变量。

```
struct Point
{
    double x;
    double y;
    double z;
};
struct Point oPoint1={100,100,0};
struct Point oPoint2;
```

其中结构体 `struct Point` 为新的数据类型, 在定义变量的时候均要有保留字 `struct`, 而不能像 `int` 和 `double` 那样直接使用 `Point` 来定义变量。此时, 就可以采用 `typedef` 将该结构体类型重新定义一个较为简洁的类型:

```
typedef struct
{
    double x;
```



```
double y;
double z;
}Point;
```

经过如上的修改后, 定义该结构体 struct Point 变量的方法就可以简化为:

```
Point oPoint;
```

3. 为数组定义简洁的类型名称

例如, 下面语句定义了三个长度为 5 的整型数组:

```
int a[10], b[10], c[10], d[10];
```

在 C++ 中, 可以将长度为 10 的整型数组看作一个新的数据类型, 再利用 typedef 为其重定义一个新的名称, 以更加简洁的形式定义此种类型的变量, 具体的处理方式如下:

```
typedef int INT_ARRAY_10[10];
typedef int INT_ARRAY_20[20];
INT_ARRAY_10 a, b, c, d;
INT_ARRAY_20 e;
```



提示 其中 INT_ARRAY_10 和 INT_ARRAY_20 为新的类型名, 10 和 20 为数组的长度。a, b, c, d 均是长度为 10 的整型数组, e 是长度为 20 的整型数组。

4. 为指针定义简洁的名称

类型重定义 typedef 首先可以为数据指针定义新的名称, 例如:

```
typedef char * STRING;
STRING csName={"Jhon"};
```

其次还可以为函数指针定义新的名称, 例如:

```
typedef int (*MyFUN)(int a, int b);
int Max(int a, int b);
MyFUN *pMyFun;
pMyFun= Max;
```

【范例 9-9】类型重定义 typedef 的应用。该范例使用类型重定义 typedef 定义了多个类型, 其在实际程序中与基本数据类型的使用方式相同, 其实现代码如代码清单 9-9 所示。

代码清单 9-9

```
1  #include <iostream.h>
2  typedef int COUNT;                //重定义基本数据类型
3  typedef double AREA;
4  typedef struct                    //重定义结构体类型
5  {
6      double x;                    //定义成员变量
7      double y;
8      double z;
9  }Point;
10 //typedef int ARRAY_10[10];        //重定义数组类型
11 //typedef int ARRAY_20[20];
12 typedef char * STRING;            //重定义指针类型
13 void main()
14 {
15     COUNT a=100;                  //用重定义的类型定义变量
16     cout<<"a= "<<a<<endl;
17     AREA b=50.02;                  //用重定义的类型定义变量
18     cout<<"b= "<<b<<endl;
19     Point op={1.0,2.0,3.0};        //用重定义的类型定义变量
20     cout<<"op.x= "<<op.x<<"\t"<<"op.y= "<<op.y<<"\t"<<"op.z= "<<op.z<<endl;
```

```

21     STRING csName={"John"};           //用重定义的类型定义变量
22     cout<<"csName= " <<csName<<endl;;
23 }

```

【运行结果】在 Visual C++中运行上述程序，其结果如图 9-13 所示。

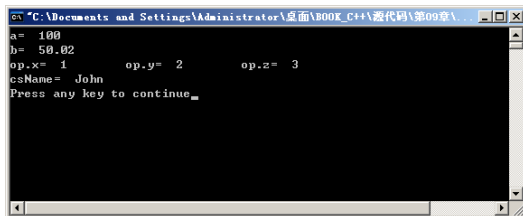


图 9-13 类型重定义

【范例解析】上述程序用 typedef 重定义了基本数据类型、结构体类型、数组类型和指针类型，并使用重定义后的关键字定义相对应的变量输出。

读者可以看到，类型重定义后就可以使用新的名称进行变量定义了。需要注意的是，在使用 typedef 时，应当注意如下事项：

- typedef 的目的是为已知数据类型增加一个新的名称，并没有引入新的数据类型。
- typedef 只适于类型名称定义，不适合变量的定义。
- typedef 与 #define 具有相似之处，但是实质不同。

比如，#define AREA double 与 typedef double AREA 可以达到相同的效果。但是其实质不同，#define 为预编译处理命令，主要定义常量，此常量可以为任何的字符及其组合，在编译之前，将此常量出现的所有位置，用其代表的字符或字符组合无条件地替换，然后进行编译。



提示 typedef 是为已知数据类型增加一个新名称，其原理与使用 int、double 等保留字一致。

9.5 位域

在实际的程序设计中，有时需要存储少量的信息，这些信息并不需要占用一个完整的字节，而只需占几个或一个二进制位。例如，在存放一个标志时，只有 0 和 1 两种状态，用一个二进制位即可。如果给其分配一个字节的存储空间，便浪费了存储空间。因此，C++引入了位域这一数据类型。

9.5.1 定义位域变量

所谓位域是把一个字节中的二进制位划分为几个不同的区域，并说明每个区域的位数。每个域有一个域名，允许在程序中按域名进行操作。这样就可以把几个不同的对象用一个字节的二进制位域来表示。位域的定义和位域变量的说明与结构定义类似，其形式如下：

```

struct 位域结构名
{
    位域列表
};

```

其中，位域列表的形式为：

类型说明符 位域名:位域长度

例如，下面语句定义了一个位域 abc，其包含 3 个位域成员 a、b 和 c。

```

struct abc
{

```



```
int a:8;
int b:2;
int c:6;
};
```

此外，位域变量的说明与结构变量说明的方式也是相同的。可采用先定义后说明、同时定义说明或者直接说明这三种方式。例如，下面语句定义了一个位域变量 `data`。

```
struct abc
{
    int a:8;
    int b:2;
    int c:6;
}data;
```

上述语句说明 `data` 为 `bs` 变量，共占两个字节。其中位域 `a` 占 8 位，位域 `b` 占 2 位，位域 `c` 占 6 位。此外，对于位域的定义有以下几点需要说明：

(1) 一个位域必须存储在同 1 个字节中，不能跨两个字节。若一个字节所剩空间不够存放另一个位域时，应从下一个字节起存放该位域。例如：

```
struct bs
{
    unsigned a:4;
    unsigned :0;           //空域
    unsigned b:4;           //从下一单元开始存放
    unsigned c:4;
}
```

在这个位域定义中，`a` 占第一字节的 4 位，后 4 位填 0 表示不使用，`b` 从第二字节开始，占用 4 位，`c` 占用 4 位。

(2) 由于位域不允许跨两个字节，因此位域的长度不能大于 1 个字节的长度，也就是说，不能超过 8 位二进制。

(3) 位域可以无位域名，这时它只用来作填充或调整位置。无名的位域是不能使用的。例如：

```
struct k
{
    int a:15;
    int :2 /*该 2 位不能使用*/
    int b:3;
    int c:2;
};
```



提示 从以上分析可以看出，位域本质上就是一种结构类型，不过其成员是按二进制分配的。

9.5.2 使用位域

由于位域本质上是一种结构类型，因此，位域的使用和结构成员变量的使用基本相同，其引用的一般形式有如下三种：

- 位域变量名.位域成员名
- 位域指针名->位域成员名
- (*位域指针名).位域成员名

【范例 9-10】位域的应用。该范例定义了位域结构 `bs`，三个位域为 `a`、`b`、`c`。说明了 `bs` 类型的变量 `bit` 和指向 `bs` 类型的指针变量 `pbit`，这表示位域也是可以使用指针，其实现代码如代码清单 9-10 所示。

代码清单 9-10

```

1  #include <iostream.h>
2  struct bs                                //定义位域结构
3  {
4      unsigned a:1;                        //定义成员变量
5      unsigned b:3;
6      unsigned c:4;
7  }bit,*pbit;                             //定义位域
8  void main()
9  {
10     bit.a=1;                             //位域初始化
11     bit.b=7;
12     bit.c=15;
13     cout<<"bit.a=  "<<bit.a<<"\t"<<"bit.b=  "<<bit.b<<"\t"<<"bit.c=  "<<bit.
c<<endl;
14     pbit=&bit;                           //指向位域的指针
15     pbit->a=0;                            //赋值
16     pbit->b&=3;                           //位域的运算
17     pbit->c|=1;
18     cout<<endl<<"bit.a=  "<<bit.a<<"\t"<<"bit.b=  "<<bit.b<<"\t"<<"bit.c=
"<<bit.c<<endl;
19 }

```

【运行结果】在 Visual C++ 中运行上述程序，其结果如图 9-14 所示。

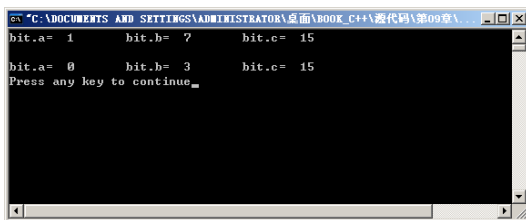


图 9-14 位域的使用

【范例解析】上述程序中，程序的 10、11、12 三行分别给三个位域赋值（应注意赋值不能超过该位域的允许范围）。程序第 13 行以整型量格式输出三个域的内容。第 14 行把位域变量 bit 的地址送给指针变量 pbit。第 15 行用指针方式给位域 a 重新赋值，赋值为 0。



注意 程序第 16 行中使用了复合位运算“&=”，相当于：pbit->c=pbit->c|1，其结果为 15，而程序第 17 行用指针方式输出了这三个域的值。

9.6 小结

本章重点讲解了 C++ 中的构造数据类型。C++ 中，结构体、共用体和枚举类型是使用较多的构造数据类型。其中，结构体类型是一种复杂而灵活的构造数据类型，它可以将多个相互关联，但类型不同的数据项作为一个整体进行处理。在定义结构体变量时，每一个成员都要分配空间存放各自的数据。共用体是另一种构造数据类型，但在定义共用体变量时，只按占用空间最大的成员来分配空间，在同一时刻只能存放一个数据成员的值。结构体和共用体变量的定义都有三种形式，可以将类型的说明和变量的定义分开、结合或不给出类型名只定义变量。读者应重点掌握结构体、共用体和枚举的变量定义和引用方法，对于位域，做简单了解即可。



9.7 习题

1. 已知有如下结构体:

```
struct st { int n; struct st *next; };
struct st a[3]={1, &a[1], 3, &a[2], 5, &a[0]}, *p;
```

如果下述语句的显示是 2, 则对 p 的赋值用什么语句来实现?

```
cout<< ++(p->next->n);
```

【解答】该习题主要考查结构体数组的使用。上述语句定义了结构体类型 st, 其包含一个整型变量和指向该类型的指针, 事实上其为一个链表节点。在声明结构体数组 a 的同时为其进行了初始化, 调用该数组元素时其结果显示 2, 表示 p->next->n 的值必须为 1, 经过前置++运算后其值才会变为 2。在初始化后只有第 1 个链表节点的 n 值为 1, 因此 p 必须指向&a[2], 正确实现语句应为 p=&a[2]。

2. 编写一个 C++ 程序, 声明一个表示时间的结构体, 可以精确表示年、月、日、小时、分、秒, 提示用户输入年、月、日、小时、分、秒的值, 然后完整地显示出来。例如, 输入 2010 5 20 18 30 30 后其输出结果如图 9-15 所示。

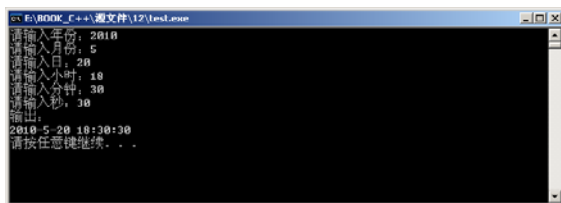


图 9-15 输出时间

【解答】该习题主要考查结构体的定义和结构体变量的声明, 及其在具体程序中引用结构体成员的方式。该习题要求先定义一个表示时间的结构体类型, 再在主程序中声明该结构体变量, 通过用户输入, 将具体数据输入到结构体成员中, 最后将结构体成员中的值输出。其简要的实现代码如下所示。

```
struct time
{
    int year;
    int month;
    int day;
    int hour;
    int minute;
    int second;
};
int main()
{
    struct time t;
    .....
    cout<<t.year<<"-"<<t.month<<"-"<<t.day<<"  "<<t.hour<<":"<<t.minute<<":"<<t.
second<<endl;
}
```

3. 编写一个 C++ 程序, 定义一个联合类型, 其成员为整型变量 i, 单精度浮点型变量 f 和双精度浮点型变量 d, 声明一个联合变量和指针, 给该联合类型的成员变量赋值, 最后输出到屏幕。

【解答】该习题主要考查联合类型的定义和联合变量的声明。在程序中首先定义该联合类型, 在主程序中声明联合变量和指针, 通过 3 种方式可引用联合类型的成员, 实现赋值和输出

功能。其简要的实现代码如下所示。

```
union variant
{
    int i;
    float f;
    double d;
};
union variant s={100};
int main()
{
    union variant *p=&s;
    cout<<"s.i= " <<s.i<<endl;
    cout<<"p->f= " <<p->f<<endl;
    cout<<"(*p).d= " <<(*p).d<<endl;
}
```

4. 设有以下说明和定义:

```
typedef union
{
    long i;
    int k[5];
    char c;
} DATE;
struct data
{
    int cat;
    DATE cow;
    double dog;
} too;
DATE max;
```

则语句 `cout<<(sizeof(struct data)+sizeof(max));` 的执行结果是多少?

【解答】该习题主要考查 struct 与 union 的区别 (一般假定在 32 位机器上)。上述语句中, DATE 是一个 union, 变量公用空间。里面最大的变量类型是 int[5], 占用 20 个字节, 所以它的大小是 20。data 是一个 struct, 每个变量分开占用空间, 依次为 $\text{int}4 + \text{DATE}20 + \text{double}8 = 32$ 。因此, 上述语句的输出结果是 $20 + 32 = 52$ 。

5. 编写一个程序, 根据用户输入的学生数目, 给出用户输入学生学号、姓名、成绩等输入提示, 用户输入完成后给出输入的各项成绩和平均分, 实现如图 9-16 所示的结果。

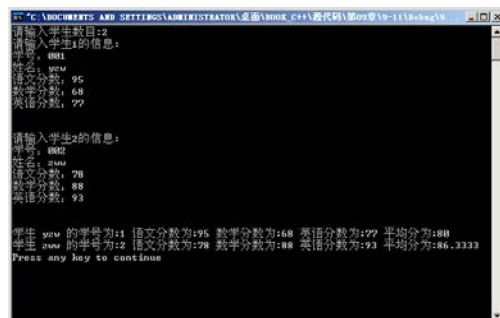


图 9-16 运行结果

【解答】该程序段需定义结构体数组变量 stu[], 其中可存储 100 个元素, 其中每个元素都包含 num、name、score 等成员变量。在主函数 main() 中, 用 for 循环控制读者输入几个学生的信息, 将这些信息分别存储在对应学生的成员变量中。在输出这些信息时, 代码中使用结构体指针指向数组元素的第一个元素依次输出。其简要的实现代码如下所示。



```

struct student                                //定义结构体
{
    int num;                                  //定义整型变量成员
    char name[10];                            //定义字符型数组成员
    float score[3];                           //定义浮点型数组成员
    float avg_score;
    student *next;                            //定义成员变量
}stu[100],*p;                                //定义结构体数组变量和结构体指针
void print(student *p)                        //定义输出函数
{
    cout<<"学生 "<<p->name<<" 的学号为:"<<p->num; //输出学生的姓名和学号
    p->avg_score=(p->score[0]+p->score[1]+p->score[2])/3;
                                                //计算平均成绩
    cout<<" 语文分数为:"<<p->score[0]<<" 数学分数为:"<<p->score[1]<<" 英语分数
为:"<<p->score[2]<<" 平均分为:"<<p->avg_score<<endl; //输出成绩
}
int main()
{
    int stu_num;                              //定义整型变量
    cout<<"请输入学生数目:";                  //输入提示
    cin>>stu_num;                              //接收键盘输入
    for(int i=0;i<stu_num;i++)                 //结构体成员初始化
    {
        cout<<"请输入学生"<<i+1<<"的信息: "<<endl<<"学号: "; cin>>stu[i].num;
        cout<<"姓名: "; cin>>stu[i].name;      //接收键盘输入
        cout<<"语文分数: "; cin>>stu[i].score[0]; //输入分数
        cout<<"数学分数: "; cin>>stu[i].score[1];
        cout<<"英语分数: "; cin>>stu[i].score[2];
        cout<<endl<<endl;                      //输出换行
    }
    for(i=0;i<stu_num;i++)                     //next 成员的初始化
        stu[i].next=&stu[i+1];
    stu[stu_num-1].next=NULL;                  //最后一个元素的 next 值为空
    p=&stu[0];                                  //指向数组变量的第一个元素
    do{
        print(p);                              //循环输出数组元素
        p=p->next;                              //指向下一个元素
    }while(p!=NULL);                            //当指针不为空时循环
    return 0;
}

```

第三篇 C++面向对象编程篇

第 10 章 类和对象

读者知道，C++是一种面向对象的程序设计语言，这是其与 C 语言的最大不同点。前面介绍了 C++作为一个面向过程的语言所具有的功能，从本章开始介绍 C++的面向对象的功能。面向对象的语言除了提供一些程序设计语言常规语法现象外，很多语法现象和概念与人类的认知规律也具有对应关系，而本章要介绍的类和对象就是其中最重要的。

以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为一个学习的参照。

- 掌握 C++中类和对象的概念。
- 掌握 C++中类的构造函数、析构函数的定义和应用。
- 掌握友元的概念和相关应用。

10.1 类

面向对象程序设计中的类，最初应该来自分析模型或者设计模型，其中对于类所具有的属性 and 操作都进行了严格定义，几乎可以将这些设计得十分详细、以图形方式描述类直接转换成 C++的类。本节介绍的是在语言中如何将类描述出来。

10.1.1 什么是类

类 (Class)，指的是具有相似内部状态和行为的实体的集合。类的概念来自于人们认识自然、认识社会的过程。在这一过程中，人们主要使用两种方法：由特殊到一般的归纳法和由一般到特殊的演绎法。在归纳的过程中，从一个个具体的事物中把共同的特征抽取出来，形成一个一般的概念，即“归类”。例如：人、狮子、老鹰等，因为其都能动，所以将其归类为动物，如图 10-1 所示。

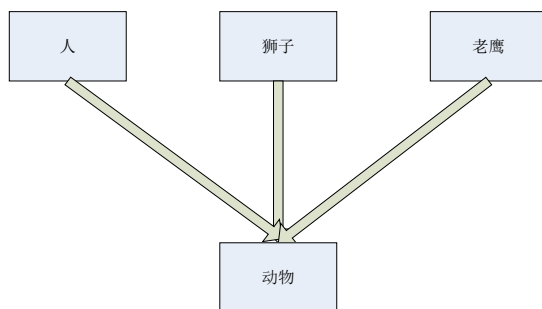


图 10-1 类的概念

在演绎的过程中，又可以将同类的事物，根据其不同的特征分成不同的小类，即“分类”。例如：动物->猫科动物->虎->东北虎等。

对于一个具体的类，其包含许多具体的个体，这就是前面所提到的“对象”。如东北虎就是一个具体的对象。由此可以看出，对象是类的一个具体的个体，而类是具有相似特征的对象集



合。相对于对象的属性和方法，类也有其内部状态和行为。类的内部状态是指类集合中对象的共有状态，类的行为是指类集合中对象的共有行为。关于类和对象的特性在后续章节中将具体讲解。



类是面向对象中最为基础的一个概念，C++中许多应用都是以类来实现的。Microsoft 为 Visual C++ 集成了许多类，称为 MFC（微软基础类库）。

10.1.2 结构到类

在前面介绍了，C++中可以定义结构体类型，将多个相关的变量包装为一个整体使用。在结构体中的变量，可以是相同、部分相同，或完全不同的数据类型。事实上，在 C++中，结构体除了可以包含变量外，还可以包含函数。

【范例 10-1】简单结构体。该范例定义了一个结构体及其变量，该结构体包含两个成员变量，在主函数中调用输出语句可以将这个结构体中定义的变量的值输出，其实现代码如代码清单 10-1 所示。

代码清单 10-1

```
1  #include <iostream.h>
2  struct point                                //定义结构体
3  {
4      int x;                                  //定义成员变量
5      int y;
6  };
7  void main()
8  {
9      point pt;                                //定义结构体变量 pt
10     pt.x=0;                                  //引用成员变量并赋值
11     pt.y=1;
12     cout<<"pt.x= " <<pt.x<<endl;           //输出
13     cout<<"pt.y= " <<pt.y<<endl;
14 }
```

【运行结果】上述代码在 Visual C++中执行，其结果如图 10-2 所示。

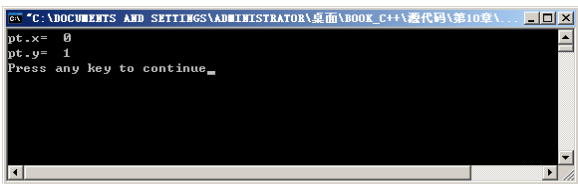


图 10-2 简单结构体的应用

【范例解析】该范例的功能是输出结构体变量内成员变量的值，在这个结构体当中，只定义了两个整型的变量作为一个点的 x 坐标和 y 坐标。在主函数 main()中，定义了一个结构体的变量 pt，对 pt 的两个成员变量进行赋值，然后调用 C++的输出流类的对象 cout 将这个点的坐标输出。



在这段程序中，结构体只包含两个变量的定义。

【范例 10-2】包含成员函数的结构体。该范例将范例 10-1 结构体的定义稍作修改，在其中增加一个输出成员变量的函数 print()，这样就不需要在 main()函数中使用 cout 语句，print()

函数称为成员函数，其实现代码如代码清单 10-2 所示。

代码清单 10-2

```

1  #include <iostream.h>
2  struct point                                //定义结构体
3  {
4      int x;                                  //定义成员变量
5      int y;
6      void print()                            //定义成员函数
7      {
8          cout<<"x= " <<x<<endl;           //输出成员变量的值
9          cout<<"y= " <<y<<endl;
10     }
11 };
12 void main()
13 {
14     point pt;                                //定义结构体变量
15     pt.x=0;                                  //引用成员变量并赋值
16     pt.y=1;
17     pt.print();                             //引用成员函数
18 }
```

【运行结果】上述代码在 Visual C++ 中执行，其结果如图 10-2 所示。

【范例解析】读者可以注意到，代码 10-2 与代码 10-1 的区别在于：代码 10-2 将输出成员变量的语句放在了结构体内部，而不是主函数 main() 中。代码 10-2 在结构体内部定义了一个没有返回值的函数 print()，而在主函数中只是调用了该函数进行输出，得到的返回结果与范例 10-1 相同。



警告 调用结构体内的成员函数，需加上后面的()括号，表示其调用的是成员函数，而不是成员变量，否则系统将给出编译错误信息。

【范例 10-3】结构体到类。该范例再对上述结构体的定义作一些修改，引出 C++ 中类的概念。范例仅仅将范例 10-2 的结构体定义关键字 struct 改为类定义的关键字 class，其他均不变，代码如代码清单 10-3 所示。

代码清单 10-3

```

1  #include <iostream.h>
2  class point                                  //声明类
3  {
4      int x;                                  //定义成员变量
5      int y;
6      void print()                            //定义成员函数
7      {
8          cout<<"x= " <<x<<endl;           //输出成员变量的值
9          cout<<"y= " <<y<<endl;
10     }
11 };
12 void main()
13 {
14     point pt;                                //声明对象
15     pt.x=0;                                  //赋值
16     pt.y=1;
17     pt.print();                             //调用成员函数
18 }
```



【运行结果】上述代码在 Visual C++ 中执行，其结果如图 10-3 所示。



图 10-3 错误提示信息

【范例解析】读者可以看到，上述程序运行后将会出现如图 10-3 所示的错误提示信息，提示不能访问类中私有（private）的成员变量和成员函数。结构体默认情况下，其成员是公有（public）的；类默认情况下，其成员是私有（private）的，这是结构体与类的区别之一。在一个类当中，公有成员是可以在类的外部进行访问，而私有成员就只能在类的内部进行访问，因此出现上述“不能访问私有成员”的错误提示信息。

事实上，如要解决上述问题，只需将类中定义的变量 *x* 和 *y*，以及定义的成员函数 *print()* 都定义为公有的就可以，即在其上加前面标识符 *public*，代码如下所示。

```
class point
{
public:                                     //说明符
    int x;
    int y;
    void print()
    {
        cout<<"x= "<<x<<endl;
        cout<<"y= "<<y<<endl;
    }
};
```



注意 读者从上述代码可以看出，类与结构体的区别除了使用关键字“class”和“struct”不同之外，更重要的是在成员的访问控制方面有所差异。

10.1.3 类的声明

经过上述内容的讲解，读者已经了解到，类描述了具有共同特征的一组对象，这组对象的属性和行为相同，只不过具体对象的属性值等有所区别。C++ 中类的定义一般分为类的声明部分和类的实现部分。其中类声明的格式如下：

```
class <ClassName>
{
private:
    私有数据和函数
public:
    受保护数据和函数
};
```

其中，参数说明如下。

- *class*：类说明的关键字。
- <ClassName> 是用户自定义的 C++ 标识符，Visual C++ 中类名的风格是，所有类的名字都以大写字母 *C* 开头，以表示这是个类的名字，例如 *CBOOK*，*CStudent* 等。
- {}：被花括号括起来的部分称为类体。类体主要由一些变量和函数说明组成，分别称为类的数据成员和函数成员，统称为类成员。



注意 与结构体的声明类似，类的声明也是以分号结束的。

【范例 10-4】类的声明。声明了一个图书类，其包含许多成员变量和成员函数，其中代码如代码清单 10-4 所示。

代码清单 10-4

```

1  class Cbook                                //声明类 Cbook
2  {
3  private:                                    //下面的为私有数据
4      char * m_pczName;
5      int m_nPages;
6      int m_nEdition;
7  public:                                     //下面的为公有数据和函数
8      void GetBookName(char *pName);
9      int GetTotalPages();
10     int GetBookEdition();
11 private:                                    //下面的为私有数据和函数
12     void SetBookName(char * pName);
13     void SetTotalPages(int nPages);
14     void SetBookEdition(int nEdition);
15 public:                                     //下面的为公有数据和函数
16     Cbook();
17 };
18 void main()
19 {
20     Cbook op1;                               //声明对象
21     cout<<"Class define Success"<<endl;    //输出信息
22 }
```

【运行结果】上述程序在 Visual C++中执行，其运行结果如图 10-4 所示。

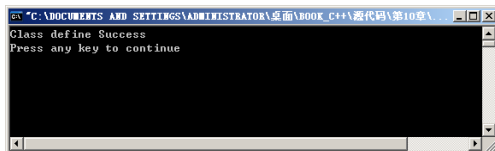


图 10-4 类的声明

【范例解析】上述代码中，声明了一个类 Cbook，该类包含了私有数据成员和公有成员，每一类中都包含变量和函数。在 main()函数中，上述程序使用该类定义了一个对象 op1。关于对象的定义，在后续章节中还将具体讲解。

Cbook 类中定义了私有和公有两类成员，其数据成员都为私有，这是出于封装的目的，不希望直接访问数据成员，而是通过所提供的公有函数访问。例如，要知道书的名字可调用函数 GetBookName()，要改变书的版本号要调用 SetBookEdition()。



提示 上述代码在编译时，Visual C++编译系统会给出“unreferenced local variable”的警告信息，这是由于该对象没有被用到，这里忽略该警告信息即可。

10.1.4 类的访问控制

前面类的声明中，定义了两个类别的成员，分别为私有成员（private）类别和公有成员（public）类别。事实上，C++中，类体被分成以下三类。



- 公有成员：以关键字 `public` 指明。
- 私有成员：以关键字 `private` 指明。
- 保护成员，以关键字 `protected` 指明。

上述的这几个关键字被称为访问说明符（`access specifier`），用来控制相应成员在程序中的可访问性，使得信息封装和模块化的风格更好。访问说明符说明了对该说明符与下一个说明符之间出现的类成员的访问限制。

关于类的上述三种成员的访问控制，在 C++ 中有一定的访问规则，如下所示：

- 对类的公有成员而言，在程序的任何位置都能够以正确的方式引用它。
- 类的私有成员只能被其自身成员所访问。即私有成员的名字只能出现在所属类类体、成员函数中，不能出现在其他函数中。
- 类的保护成员只能在该类的派生类类体中使用。

关于类的保护成员的使用，将在后续章节关于类的继承中详细讲解。此外，类的友元可以访问类的任何成员，这也将后续章节中提到。

【范例 10-5】类的访问控制。该范例声明了一个类，其包含私有成员和公有成员，在主函数中对该类的私有成员和公有成员进行访问，读者可查看其访问结果，其实现代码如代码清单 10-5 所示。

代码清单 10-5

1	#include <iostream.h>	
2	class point	//声明类
3	{	
4	private:	
5	int x;	//定义成员变量
6	int y;	
7	public:	
8	void print()	//定义成员函数
9	{	
10	cout<<"x= "<<x<<endl;	//在类中访问类的私有成员
11	cout<<"y= "<<y<<endl;	
12	}	
13	};	
14	void main()	
15	{	
16	point pt;	//声明对象
17	pt.x=0;	//在外部访问类的私有成员（错误）
18	pt.print();	//访问类的公有成员
19	}	

【运行结果】在 Visual C++ 中新建一个 **【C++ Source File】**，输入如上代码后编译，系统会给出如图 10-5 的编译错误信息。

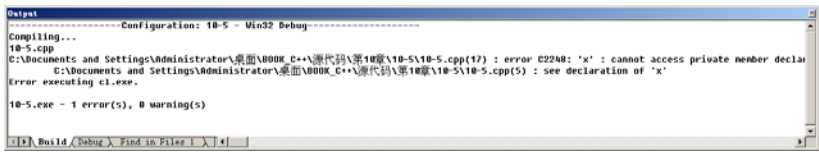


图 10-5 编译错误信息

【范例解析】上述代码中，类 `point` 中定义了 `x` 和 `y` 为私有成员，函数 `print()` 为公有成员。在上述代码第 17 行调用了私有成员 `x`，这是不符合类的访问控制规则的，所以出现了如图 10-5 的编译错误，该错误表示不能访问私有成员。而第 18 行 `pt.print()` 调用的是公有成员，这是允

许的, 因此, 将第 17 行注释去掉就可以通过编译了, 其执行结果如图 10-6 所示。

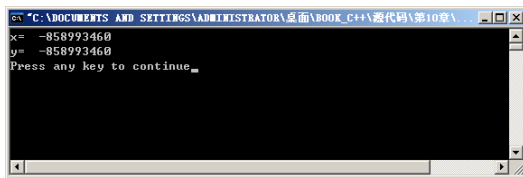


图 10-6 执行结果

由于没有给类的成员 `x` 和 `y` 赋值, 因此, 在主函数 `main()` 中调用 `print()` 函数输出 `x` 和 `y` 的值时, 都会输出没有意义的值, 如图 10-6 所示。



注意 类体中可以出现多个说明符, 每个说明符也可出现多次, 不同说明符的出现次序没有限制。当从类体开始到某些类成员前没有访问说明符, 或类体中根本没有访问说明符时, 这些成员被默认是私有成员。

例如, 下列类的声明:

```
class point                                //定义类
{
    int x;                                //定义成员变量
    int y;
public:
    void print()                            //定义成员函数
    {
        cout<<"x= " <<x<<endl;
        cout<<"y= " <<y<<endl;
    }
};
```

与代码 10-5 中类的声明是相同的, 虽然其没有 `private` 标识符定义 `x` 和 `y` 为私有成员, 但类将这些成员默认为私有的, 这是为了保护数据而设计的。

10.1.5 类的定义

前面章节讲解的类的说明声明了类的内部结构(数据成员)及类的接口(成员函数的函数原型), 但这些函数的功能的实现并没有进行具体定义, 因此要给出这一类对象的具体行为还应该对类的成员函数进行定义, 即类的实现。

由于数据成员作为类体的部分已经在类说明中被说明和定义(对于常量对象), 而类体的另一部分是成员函数, 因此成员函数的定义有时也被称为类体的定义或类的定义。一般来说, 成员函数的定义格式如下:

```
<ReturnType><ClassName>::<FunctionName>(<ArgumentList>)
{
    函数体
}
```

其中, 参数说明如下。

- `<ReturnType>`: 表示该成员函数的返回类型, 其可以是基本数据类型和构造数据类型。
- `<ClassName>`: 表示类的名称。如果在类的内部定义成员函数, 则类的名称可以省略。
- `::`: 域运算符, 其用来指定后面的函数是属于哪一个类。同样, 如果在类的内部定义成员函数, 该运算符和类名可一起省略。
- `<FunctionName>`: 函数名, 其可以为任一标识符。



- <ArgumentList>: 参数列表。

读者可以看出，成员函数与一般函数定义不同的是多了类名（<ClassName>）和域运算符（::），它们是用来指明所定义的函数属于哪个类，在类外定义成员函数的话这是必需的。因为不同类中的成员函数可能重名，这种情况在客观世界中十分常见，不指定成员函数所属的类就无法知道定义的成员函数是哪一个类的。

花括号括起来的部分是函数体，其属于类体的一部分，其中可以直接调用类的所有成员，就如同它们是在函数体内定义的自动变量一样，包括数据成员和成员函数，因此不用再显式地指明所属类体。



注意 在函数中所调用的属性和其他成员函数为当前对象所具有的属性和成员函数。

【范例 10-6】 定义类的成员函数的实现。该范例定义了代码 10-4 所声明的 Cbook 类中的几个成员函数，读者可以仔细理解定义成员函数的格式，其实现代码如代码清单 10-6 所示。

代码清单 10-6

```

1  #include <iostream.h>
2  #include <string.h>
3  class Cbook                                //声明类 Cbook
4  {
5  private:                                    //下面的为私有数据
6      char * m_pczName;
7      int m_nPages;
8      int m_nEdition;
9  public:                                     //下面的为公有数据和函数
10     void GetBookName(char * pName);
11     int GetTotalPages();
12     int GetBookEdition();
13 private:                                    //下面的为私有数据和函数
14     void SetBookName(char * pName);
15     void SetTotalPages(int nPages);
16     void SetBookEdition(int nEdition);
17 public:                                     //下面的为公有数据和函数
18     Cbook();
19 };
20 void Cbook::GetBookName(char * pName)        //定义成员函数
21 {
22     strcpy(pName, m_pczName);
23 }
24 int Cbook::GetBookEdition()                  //定义成员函数
25 {
26     return m_nEdition;
27 }
28 void Cbook::SetBookName(char * pName)        //定义成员函数
29 {
30     if(m_pczName!=0)
31         delete[] m_pczName;                  //如果已经有了旧名字，删除它，然后
                                                //重新命名
32     m_pczName=new char[strlen(pName)+1];      //重新分配存储空间
33     strcpy(m_pczName, pName);                 //复制字符串
34 }
35 void Cbook::SetTotalPages(int nPages)        //定义成员函数
36 {
37     m_nPages=nPages; }
38 void Cbook::SetBookEdition(int nEdition)    //定义成员函数

```

```

39 {
40     m_nEdition=nEdition;
41 }
42 void main()
43 {
44     Cbook opl;                //声明该类的对象
45     int i;
46     i=opl.GetBookEdition();    //调用成员函数
47     cout<<i<<endl;
48 }

```

【运行结果】在 Visual C++ 中新建一个【C++ Source File】，输入如上代码后编译，若编译无误后运行，其结果如图 10-7 所示。

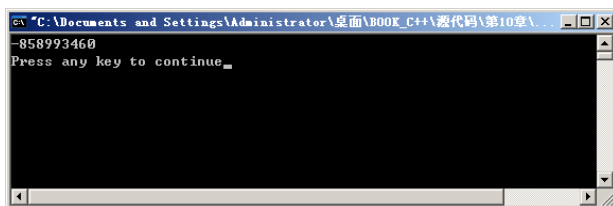


图 10-7 定义成员函数的执行结果

【范例解析】上述代码中，首先声明了类 Cbook，接着定义了该类声明的成员函数，然后在主函数 main() 中声明了一个该类的对象，最后调用了其中的一个成员函数 GetBookEdition()，得到如图 10-7 的输出结果。



提示 上述范例中，调用的 GetBookEdition() 函数返回的是成员变量 m_nEdition 的值，由于 m_nEdition 没有进行初始化，因此其返回值为一个无意义的随机数。

此外，类的说明通常放在一个以 .h 为扩展名的文件中，称为头文件，其中定义了类的接口 (interface)，因此可以同其他类的说明同放于一个文件。

如果类说明的程序行较多，那么应该将它放在一个独立文件中，Visual C++ 的风格是以主文件名作为类名去掉前面的字符 C，例如 Cbook 的类说明可以放在文件 book.h 中，将类体的定义放于一个以 .cpp 为扩展名的文件中，这称为类的实现文件。在这个文件的开始部分应该用文件包含指令将类说明文件包含进来。

10.2 对象

10.1 节内容讲解了类的概念及其在 C++ 中的应用，事实上，在 C++ 中具体的应用都是对于一个个具体的对象。10.1 节的代码中也出现过对象的声明，本节将重点介绍面向对象中对象的概念和在 C++ 中的一些应用。

10.2.1 对象概述

现实世界中的对象是人们认识世界的基本单元，世界就是由这些基本单元——对象组成的。如一个人、一辆车、一次购物、一次演出等。对象可以很简单，也可以很复杂，复杂的对象可由若干个简单对象组成。对象是现实世界中的实体，其一般具有以下特性：

- 每个对象都有一个用于与其他对象相区别的名字。
- 具有某些特征，我们称它为属性或状态。
- 有一组操作，每一个操作决定对象的一种行为（即对象）能干什么。



- 对象的状态只能被自身的行为所改变。
- 对象之间以消息传递的方式相互通信。

提示 上述对象的这些特性在面向对象程序设计中分别抽象为对象名、属性、事件、方法和消息，这是面向对象程序设计的基础。

在现实世界中，对象指的就是具体的事物，例如：飞机、汽车、人等。每个对象都含有自己的内部状态和行为，如人具有名字、身高、体重等内部状态，也具有走路、吃饭等行为。在面向对象的概念中，将对象的内部状态称为属性，将其行为称为方法或事件。对象之间的联系通过消息来传递，消息机制是对象间相互联系和相互作用的方式。

简单来说，对象与类的关系是具体与抽象的关系，比如果为一个类，那么苹果就是一个对象，是水果类中的一个具体对象，如图 10-8 所示。

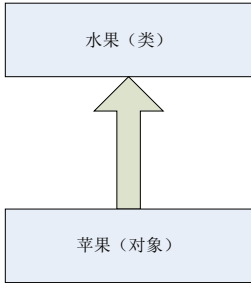


图 10-8 对象与类的关系

10.2.2 对象数组

对象数组是指以数组元素为对象的数组。该数组中若干个元素必须是同一个类的若干个对象。对象数组的定义、赋值和引用与普通数组一样，只是数组的元素与普通数组不同，它是同类的若干个对象。比如，张三、李四、王五都是人，那么其组成的数组即为对象数组，如图 10-9 所示。

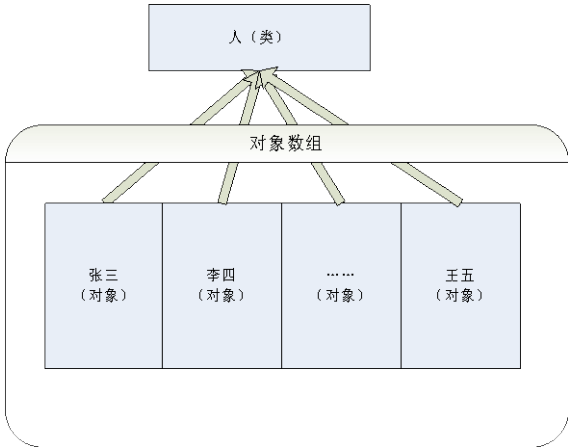


图 10-9 对象数组

一般来说，对象数组的定义格式如下：

<类名><数组名>[<大小>]...

其中，参数说明如下。

- <类名>: 指出该数组元素是属于该类的对象。
- <数组名>: 定义该对象数组的名称, 它必须是 C++ 的标识符。
- <大小>: 方括号内的<大小>给出某一维的元素个数。一维对象数组只有一个方括号, 二维对象数组要有两个方括号等。

例如, 下列语句定义了一个一维的对象数组:

```
Cbook dates[7];
```

上述语句表明 `dates` 是一维对象数组名, 该数组有 7 个元素, 其中的对象属于 `Cbook` 类, 也即每个元素都是类 `Cbook` 的对象。与普通的数组类似, 对象数组可以被赋初值, 也可以被赋值。

【范例 10-7】对象数组的定义和赋值。该范例实现了对数组对象赋值, 其实现代码如代码清单 10-7 所示。

代码清单 10-7

```

1  #include <iostream.h>
2  class DATE                                //声明类
3  {
4  public:                                    //公有成员
5      DATE(int m,int d,int y);
6      void print();
7  private:                                  //私有成员
8      int month, day, year;
9  };
10 DATE dates[4]={DATE(7,7,2001),DATE(7,8,2001),DATE(7,9,2001),DATE(7,10,2001)};
    //初始化
11 DATE::DATE(int m,int d,int y)             //定义成员函数(构造函数)
12 {
13     month=m;                               //定义成员变量
14     day=d;
15     year=y;
16 }
17 void DATE::print()                        //定义成员函数
18 {
19     cout<<month<<"\t"<<day<<"\t"<<year<<endl; //输出成员变量的值
20 }
21 void main()
22 {
23     int i;                                  //定义整型变量
24     for(i=0;i<4;i++)
25         dates[i].print();                  //调用对象的print()成员函数
26 }
```

【运行结果】在 Visual C++ 中运行上述程序, 返回的是对象数组所赋的初值, 结果如图 10-10 所示。

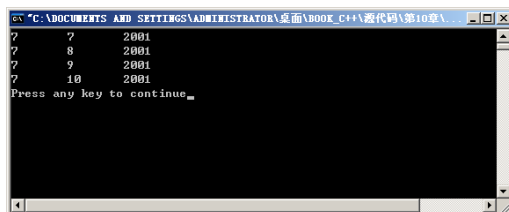


图 10-10 对象数组的定义和赋值

【范例解析】上述程序中, 首先声明了类 `DATE`, 其包含 2 个公有成员函数和 3 个私有成员



员变量，接下来定义了包含 4 个元素的对象数组 `dates[4]`，并对其初始化。函数 `print()` 将成员变量输出。在主函数 `main()` 中采用一个 `for` 循环，调用对象的 `print` 函数将初始化后的对象输出。



提示 上述代码中声明类后对其中的成员函数进行了定义，其中函数 `DATE` 是该类的构造函数，对对象进行初始化，关于构造函数在后续章节中将详细讲解。

10.3 构造函数

从代码 10-7 中，读者可以看到，在定义了类的对象数组后进行了初始化，而初始化采用了函数 `DATE()` 来实现。事实上，在定义类后，经常需要一一为对象的成员变量指定初始值，因此，C++ 中引入了构造函数的概念。

10.3.1 构造函数的概念

前面提到了，在类的使用过程中，读者可能会有这样的经历，定义类具有很多的成员变量，而又不得不由此类创建多个该类的对象。那么，当需要为每个对象的成员变量赋初值的时候，就要一一地为对象的成员变量指定初始值，这项工作是非常烦琐的。此外，在类中定义成员变量时，不能给这些变量赋初值。例如：

```
class A
{
    int x=0;    //错误，此处不能给变量 x 赋值。
};
```

上述赋值语句是错误的，它不能在类定义时给成员变量赋初值。因此，在 C++ 中，通常使用构造函数来解决这个问题。

构造函数（Constructor）是在类中定义的一种特殊的函数，其函数的名称与类的名称相同。构造函数的主要功能是为对象分配空间，也可用来为类成员变量赋初值，因此构造函数不能有返回类型，甚至不能有 `return` 语句。构造函数相当于要停车就必须找个车位来放车，这个寻找车位的操作就是构造函数所要进行的操作，如图 10-11 所示。

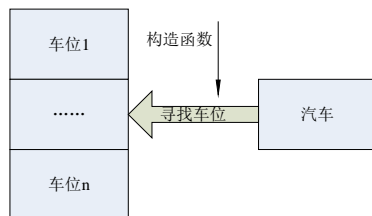


图 10-11 构造函数概念

在代码 10-7 中，声明的公有成员函数 `DATE(int m,int d,int y)` 就是一个构造函数，其功能是实现对私有成员变量 `month`、`day` 和 `year` 进行赋值，读者可以在代码 10-7 的第 11~16 行关于 `DATE` 函数定义的代码中看出。此外，构造函数还有如下的性质，这是读者需要注意的：

- 构造函数的名字必须与类的名字相同。
- 构造函数的参数可以是任何数据类型，但它没有返回值，不能为它定义返回类型，包括 `void` 型在内。
- 对象定义时，编译系统会自动地调用构造函数完成对象内存空间的分配和初始化工作。
- 构造函数是类的成员函数，具有一般成员函数的所有性质，可访问类的所有成员，可以是内联函数，可以带有参数表，可以带有默认的形参值，还可以重载。

10.3.2 构造函数的声明和定义

构造函数的声明和定义与普通成员函数的声明和定义类似。例如，下面语句给出了一个类的声明和定义，其中包含了构造函数的声明定义。

```
class complex
```

```

{
private:
    double real, imag;                //定义复数的实部和虚部
public:
    complex(double r, double i)       //定义构造函数，它的名字与类名相同
    {
        real=r;
        imag=i;
    }
    void disp()
    {
        cout<<real<<" "<<imag<<"i"<<endl;
    }
};

```

上述语句定义了一个类 `complex`，其包含两个私有成员变量，在公有成员中，其包含了两个成员函数：`complex()`函数和 `disp()`函数。其中，`complex()`函数为构造函数，其函数名与类名相同，并且带有两个参数。此外，构造函数也可以在类中声明，在外部定义。例如，将上述声明定义的类改写成如下形式，请读者观察其格式的不同。

```

class complex
{
private:
    double real, imag;                //定义复数的实部和虚部
public:
    complex(double r, double i);       //声明构造函数，其名字与类名相同
    void disp()
    {
        cout<<real<<" "<<imag<<"i"<<endl;
    }
};
complex::complex(double r, double i)  //定义构造函数，它的名字与类名相同
{
    real=r;
    imag=i;
}

```

读者可以看到，上述两种方式都声明和定义了构造函数。其中，第一种方式是在定义类的同时在类的内部定义了构造函数，而另一种方式是在类中只声明构造函数，在类的外部才定义构造函数的具体功能。



警告 在类的外部定义构造函数时，需要在函数前增加类名和域运算符，否则在编译时系统不能识别其属于哪个类。

实际应用中，一般都要给类定义构造函数，如果没有定义，编译系统就自动生成一个默认的构造函数，这个默认的构造函数不带任何参数，只能给对象开辟一个存储空间，而不能为对象中的数据成员赋初值。此时数据成员的值是随机的，程序运行时可能会出错。因此，给对象赋初值是非常重要的工作。系统自动生成的构造函数的形式为：

```

类名::构造函数名()
{
}

```

其中，构造函数名与类名同名。例如，编译系统为类 `complex` 自动生成的构造函数是：

```

complex::complex()
{
}

```



10.3.3 构造函数的调用

与其他的成员函数类似，定义了构造函数后，就可以调用了。一般来说，在定义对象的同时调用构造函数，其调用格式为：

类名 对象名(实参表)



构造函数一般不需要用户显式调用，其在声明对象时系统会自动调用构造函数。

【范例 10-8】构造函数的调用。该范例对上述定义的构造函数 `complex()` 进行调用，读者可观察该构造函数在何时何处被调用了，其调用结果是什么，代码如代码清单 10-8 所示。

代码清单 10-8

```
1  #include <iostream.h>
2  class complex
3  {
4  private:
5      double real, imag;           //定义复数的实部和虚部
6  public:
7      complex(double r, double i); //声明构造函数，其名字与类名相同
8      void disp()                  //声明成员函数
9      {
10         cout<<real<<"+"<<imag<<"i"<<endl; //输出
11     }
12 };
13 complex::complex(double r, double i) //定义构造函数，它的名字与类名相同
14 {
15     real=r;                       //成员变量初始化
16     imag=i;
17 }                                  //初始化私有数据成员 real 和 imag
18 void main()
19 {
20     complex op1(1.5,3.0);          //创建对象
21     op1.disp();                    //调用类的成员函数
22 }
```

【运行结果】在 Visual C++ 中执行上述程序，其结果如图 10-12 所示。

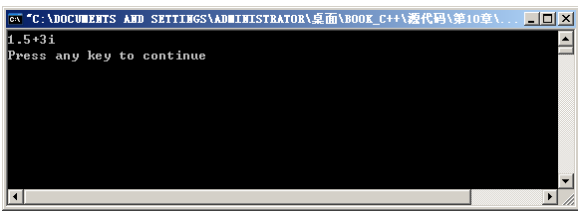


图 10-12 构造函数的调用

【范例解析】上述代码中，首先声明了类 `complex` 及其构造函数，在类外定义了该构造函数。在主函数 `main()` 中，没有直接调用该构造函数 `complex()`，而只是声明了对象 `op1`，系统自动调用了构造函数，并完成了初始化的功能。在调用了该对象的 `disp()` 函数后，将该对象建立后的结果输出，于是得到了如图 10-12 所示的结果。

10.3.4 不带参数的构造函数

通过 10.3.3 节的范例，读者可以看出，构造函数一般需要完成对类中私有成员变量的初始

化, 因此它需要包含参数, 以便在声明对象时完成对象的初始化。事实上, 构造函数可以不带参数。

【范例 10-9】不带参数的构造函数。该范例定义了一个不带参数的构造函数, 读者可理解其完成的功能与带参数的构造函数有何不同, 其代码如代码清单 10-9 所示。

代码清单 10-9

```

1  #include <iostream.h>
2  class myclass                                //定义类
3  {
4  private:                                     //定义私有成员
5      int a;
6  public:
7      myclass();                               //声明不带参数的构造函数
8      void disp()                             //定义成员函数
9      {
10         cout<<"a*a="<<a*a<<endl;           //输出
11     }
12 };
13 myclass::myclass()                          //定义构造函数
14 {
15     cout<<"initialized\n";
16     a=10;                                    //给变量初始化
17 }
18 void main()
19 {
20     myclass s;                                //创建对象, 执行构造函数
21     s.disp();                                //调用成员函数
22 }

```

【运行结果】在 Visual C++ 中执行上述程序, 其结果如图 10-13 所示。

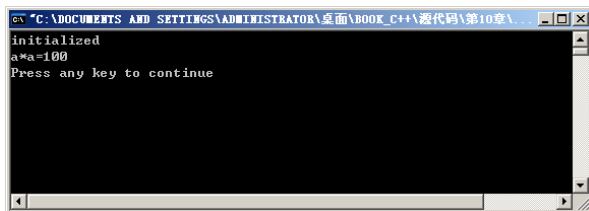


图 10-13 不带参数的构造函数

【范例解析】上述代码中, 定义了一个类 myclass, 在类中声明了该类的构造函数 myclass(), 需要注意的是, 该构造函数没有参数, 而是在类外定义该构造函数, 输出提示信息 “initialized” 并给私有变量 a 赋初值 10。在主函数 main() 中声明对象 s, 与范例 10-8 不同的是, 对象 s 在声明时没有参数。调用该对象的成员函数 disp() 后, 其运行结果如图 10-13 所示。



注意 不带参数的构造函数对象的初始化是固定的, 如希望在建立对象时通过参数初始化数据成员, 应使用带参数的构造函数。

10.3.5 带有默认参数的构造函数

当构造函数带有参数时, 在定义对象时必须给构造函数传递参数, 否则, 构造函数将不被执行。但在实际应用中, 有些构造函数的参数值通常是不变的, 只有在特殊情况下才需要改变它的值。这时, 可以将构造函数定义成带默认参数的值的构造函数, 这样, 在定义对象时可以



不指定实参，用默认参数值来初始化数据成员。

【范例 10-10】带有默认参数的构造函数。该范例定义了一个类 `complex`，其中定义了带默认参数的构造函数，其代码如代码清单 10-10 所示。

代码清单 10-10

```
1  #include <iostream.h>
2  #include <math.h>                                //调用头文件
3  class complex                                    //定义类
4  {
5      double real,imag;                            //定义私有成员变量
6  public:
7      complex(double real=0.0, double imag=0.0); //声明带有默认参数的构造函数
8      double abscomplex();
9  };
10 complex::complex(double r, double i)             //定义构造函数
11 {
12     real=r;                                       //成员初始化
13     imag=i;
14 }
15 double complex::abscomplex()                    //定义成员函数
16 {
17     double n;
18     n=real*real+imag*imag;
19     return sqrt(n);                             //返回平方根值
20 }
21 void main()
22 {
23     complex ob1;                                 //创建对象 ob1
24     complex ob2(1.1);                           //创建对象 ob2
25     complex ob3(1.1,2.2);                       //创建对象 ob3
26     cout<<"abs of complex ob1="<<ob1.abscomplex()<<endl;
27     cout<<"abs of complex ob2="<<ob2.abscomplex()<<endl;
28     cout<<"abs of complex ob3="<<ob3.abscomplex()<<endl;
29 }
```

【运行结果】在 Visual C++ 中执行上述代码，其运行结果如图 10-14 所示。

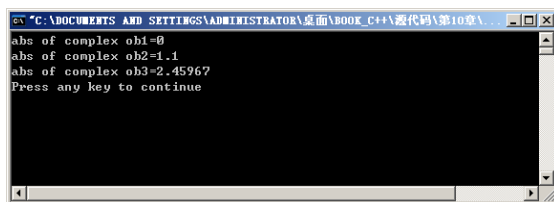


图 10-14 带有默认参数的构造函数

【范例解析】上述代码中，对象 `ob1` 在定义时没有传递参数，所以 `real` 和 `imag` 均取构造函数的默认值初始化，`real` 和 `imag` 均为 0.0。对象 `ob2` 在定义时传递了一个参数，按顺序传递给了第一个形参 `real`，第二个形参 `imag` 取默认值，所以 `real` 为 1.1，而 `imag` 为 0.0。对象 `ob3` 在定义时传递了两个参数，所以 `real` 为 1.1，`imag` 为 2.2。



提示 创建对象调用的构造函数是否带参数，其创建的对象对成员变量的初始化程度是不一样的。

10.3.6 构造函数的重载

在前面章节中介绍了 C++ 支持函数的重载, 所以一个类中也可以有多个不同参数形式的构造函数。用类去定义一个变量 (后面可以附带参数), 也就是在内存中产生一个类的实例 (用类定义的实例变量通常也叫对象) 时, 程序将根据参数自动调用该类中对应的构造函数。

事实上, C++ 允许对构造函数重载, 即可以定义多个参数及参数类型不同的构造函数, 用多种方法对对象初始化。对构造函数进行重载可以适应不同的情况, 增加程序设计的灵活性, 这些构造函数之间通过参数的个数或类型来区分。

【范例 10-11】构造函数的重载。该范例重载了构造函数, 重载后的构造函数带有两个参数, 其实现代码如代码清单 10-11 所示。

代码清单 10-11

```

1  #include<iostream.h>
2  class point                                //定义类
3  {
4  private:
5      double fx,fy;                          //定义私有成员
6  public:
7      point();                               //声明不带参数的构造函数
8      point(double x,double y);             //声明带两个参数的构造函数
9      void showpoint();
10 };
11 point::point()                             //定义不带参数的构造函数
12 {
13     fx=0.0;                                //成员初始化
14     fy=0.0;
15 }
16 point::point(double x,double y=5.5)        //定义带两个参数的构造函数
17 {
18     fx=x;                                  //成员初始化
19     fy=y;
20 }
21 void point::showpoint()                    //定义成员函数
22 {
23     cout<<fx<<" "<<fy<<endl;             //输出
24 }
25 void main()
26 {
27     point p1;                              //用构造函数 point() 创建对象
28     cout<<"the fx and fy of p1: ";
29     p1.showpoint();
30     point p2(10);                          //构造函数 point(double x,double
                                           //y=5.5) 创建对象
                                           //fy 用默认值 5.5 初始化
31
32     cout<<"the fx and fy of p2: ";
33     p2.showpoint();
34     point p3(1.1,2.0);                     //构造函数 point(double
                                           //x,double y=5.5) 创建对象
                                           //fy 被重新赋值为 2.0
35
36     cout<<"the fx and fy of p3: ";
37     p3.showpoint();                        //调用成员函数
38 }

```

【运行结果】在 Visual C++ 中执行上述代码, 其运行结果如图 10-15 所示。

【范例解析】上述代码中, 类 point 中声明了两个构造函数: point() 和 point(double x,double y),



其中前者为不带参数的，后者带两个参数。在主函数 `main()` 中声明对象时，系统自动调用对应的构造函数，如在 `point p1`；语句声明 `p1` 对象时，其调用的是 `point()` 构造函数；在 `point p2(10)`；语句声明 `p2` 对象时，其调用的是 `point(double x,double y)` 构造函数，这时只有一个实参，则另一个 `y` 为默认值 5.5；在语句 `point p3(1.1,2.0)`；中调用的是 `point(double x,double y)` 构造函数，其 `y` 的默认值被重新赋值为 2.0。分别调用 `showpoint()` 函数输出这些对象的 `fx` 和 `fy` 的值，就得到了上述结果。

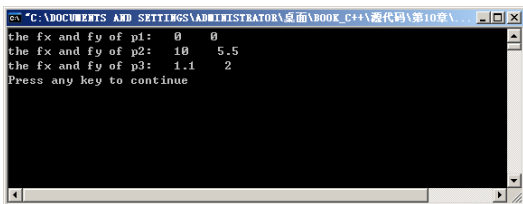


图 10-15 构造函数的重载



注意 当定义带参数的构造函数时采用了默认的参数，在调用时若指定了参数值，则使用该指定值，否则采用默认值。

10.4 拷贝构造函数

C++中，除了普通的构造函数外，还有一类特殊的构造函数——拷贝构造函数。拷贝构造函数的作用是用一个已经存在的对象来初始化该类的新对象，用户可根据需要定义拷贝构造函数，也可由系统生成一个默认的拷贝构造函数。

10.4.1 定义拷贝构造函数

由于拷贝构造函数是一种特殊的构造函数，因此其声明和定义与普通的构造函数有些区别。一般来说，自定义拷贝构造函数的形式为：

```
类名(类名&对象名)
{
    拷贝构造函数的函数体
}
```

其中，对象名是用来初始化另一个对象的引用。

【范例 10-12】自定义拷贝构造函数。该范例介绍自定义拷贝构造函数的定义，代码如代码清单 10-12 所示。

代码清单 10-12

```
1  #include<iostream.h>
2  class point                                //定义类
3  {
4  private:
5      double fx,fy;                          //定义私有成员
6  public:
7      point(point &p);                        //声明拷贝构造函数
8      point(double x,double y);              //声明构造函数
9      void showpoint();
10 };
11 point::point(point &p)                      //定义拷贝构造函数
```

```

12 {
13     fx=p.fx+10;                //定义拷贝构造函数的功能
14     fy=p.fy+20;
15 }
16 point::point(double x,double y)    //定义构造函数
17 {
18     fx=x;
19     fy=y;
20 }
21 void point::showpoint()           //定义成员函数
22 {
23     cout<<fx<<"    "<<fy<<endl;
24 }
25 void main()
26 {
27     point p1(1.1,2.2);           //用构造函数创建对象 p1
28     cout<<"the fx and fy of p1:  ";
29     p1.showpoint();              //调用类的成员函数
30     point p2(p1);                //用默认拷贝构造函数创建对象 p2
31     cout<<"the fx and fy of p2:  ";
32     p2.showpoint();              //调用类的成员函数
33 }

```

【运行结果】读者可以将代码 10-12 与代码 10-11 比较, 该代码只是将原型不带参数的构造函数定义改成了拷贝构造函数的定义, 其运行结果如图 10-16 所示。

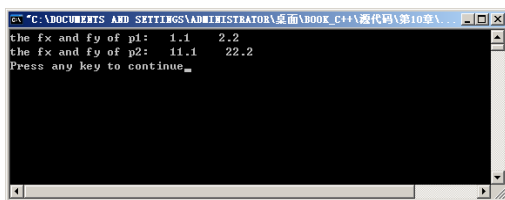


图 10-16 拷贝构造函数的定义

【范例解析】上述代码中, 在代码的第 7 行声明了一个拷贝构造函数, 在第 11~15 行定义了该拷贝构造函数的功能。在主函数 main()中, 首先 point p1(1.1,2.2);语句创建了对象 p1, 其调用的是普通的构造函数 point(double x,double y)。在代码的第 30 行使用 point p2(p1);语句创建了对象 p2, 其调用的是拷贝构造函数 point(point &p)。



注意 在创建对象时, 调用的是构造函数还是拷贝构造函数, 这是由编译系统根据需要创建对象的参数来确定的。

10.4.2 调用拷贝构造函数

10.4.1 节中在创建对象的时候系统自动调用了拷贝构造函数, 其在一个对象的基础上再创建另一个对象。一般来说, 以下三种情况拷贝构造函数会被调用:

- 用类的对象去初始化该类的另一个对象时。
- 函数的形参是类的对象, 调用函数进行形参和实参的结合时。
- 函数的返回值是类的对象, 函数执行完返回调用者时。

【范例 10-13】多种形式的拷贝构造函数的调用。该范例定义了构造函数和拷贝构造函数, 并在主函数中使用上述三种方式调用拷贝构造函数, 其实现代码如代码清单 10-13 所示。



代码清单 10-13

```

1  #include <iostream.h>
2  class point                                //定义类
3  {
4  private:                                  //定义私有成员
5      int x,y;
6  public:
7      point(int a=0,int b=0)                //定义构造函数
8      {
9          x=a;                              //初始化成员变量
10         y=b;
11     }
12     point(point &p);                        //声明拷贝构造函数
13     int getx()                             //定义成员函数
14     {
15         return x;                          //取成员变量 x 的值
16     }
17     int gety()
18     {
19         return y;                          //取成员变量 y 的值
20     }
21 };
22 point::point(point &p)                    //定义拷贝构造函数
23 {
24     x=p.x+10;                              //拷贝构造函数的功能
25     y=p.y+20;
26     cout<<"调用拷贝构造函数"<<endl;
27 }
28 void f(point p)                            //定义成员函数
29 {
30     cout<<p.getx()<<"    "<<p.gety()<<endl;
31 }
32 point g()                                 //定义成员函数
33 {
34     point q(3,5);                          //调用拷贝构造函数
35     return q;
36 }
37 void main()
38 {
39     point p1(2,4);                          //用构造函数创建对象 p1
40     point p2(p1);                          //调用拷贝构造函数的第一种情况
41     cout<<p2.getx()<<"    "<<p2.gety()<<endl;
42     f(p2);                                  //调用拷贝构造函数的第二种情况
43     p2=g();                                 //调用拷贝构造函数的第三种情况
44     cout<<p2.getx()<<"    "<<p2.gety()<<endl;
45 }

```

【运行结果】在 Visual C++ 中执行上述代码，其结果如图 10-17 所示。

图 10-17 拷贝构造函数的调用

【范例解析】上述代码中,定义了类 `point` 和拷贝构造函数 `point(point &p)`,在主函数 `main()` 中,首先用构造函数声明了对象 `p1`,接着分别采用上述三种形式调用拷贝构造函数声明了对象 `p2`,并输出该对象的 `x` 和 `y` 值。读者可根据查看其取值的不同,来分析其执行。



提示 构造函数可以在成员函数中被调用,成员函数中可以返回类类型的值,如上述代码中的第 32~36 行代码所示。

10.4.3 默认拷贝构造函数

当用一个已经存在的对象初始化本类的新对象时,如果没有自定义拷贝构造函数,则系统会自动生成一个默认的拷贝构造函数来完成初始化的工作。

【范例 10-14】默认拷贝构造函数的应用。该范例定义了类 `point`,其中定义了其默认拷贝构造函数,实现代码如代码清单 10-14 所示。

代码清单 10-14

```

1  #include <iostream.h>
2  class point                                //定义类
3  {
4  private:
5      double fx,fy;                          //定义私有成员
6  public:
7      point(double x,double y);              //声明拷贝构造函数
8      void showpoint();                      //声明成员变量
9  };
10 point ::point(double x,double y)           //定义构造函数
11 {
12     fx=x;                                  //初始化成员变量
13     fy=y;
14 }
15 void point::showpoint()                   //定义成员函数
16 {
17     cout<<fx<<"    "<<fy<<endl;
18 }
19 void main()
20 {
21     point p1(1.1,2.2);                     //用构造函数创建对象 p1
22     cout<<"the fx and fy of p1: "<<endl;
23     p1.showpoint();                         //调用成员函数
24     point p2(p1);                          //用默认构造函数创建对象 p2
25     cout<<"the fx and fy of p2: "<<endl;
26     p2.showpoint();                        //调用成员函数
27 }
```

【运行结果】在 Visual C++ 中执行上述代码,其结果如图 10-18 所示。

图 10-18 默认拷贝构造函数



【执行结果】上述代码中，定义了构造函数 `point(double x, double y)`，在主函数 `main()` 中创建对象 `p1` 时系统自动调用该函数，而在创建对象 `p2` 时，Visual C++ 编译系统则自动调用默认的构造函数来实现 `p2` 的创建。



注意 与拷贝构造函数的调用类似，默认的构造函数的调用也是由编译系统根据对象的特征自动调用的。

10.5 析构函数

与构造函数类似的，析构函数也是一种特殊的成员函数，也被声明为公有成员。不同的是，析构函数作用时释放分配给对象的内存空间，并做一些善后工作。析构函数在声明定义和使用的时候需要注意如下的事项：

- 析构函数的名字必须与类名相同，但在名字的前面要加波浪号（“~”）。
- 析构函数没有参数，没有返回值，不能重载，在一个类中只能有一个析构函数。
- 当撤销对象时，系统会自动调用析构函数完成空间的释放和善后工作。

【范例 10-15】析构函数的声明和使用。该范例是一个简单析构函数的声明，读者可以根据该范例来理解析构函数在应用程序中的使用方法和应用目的，代码如代码清单 10-15 所示。

代码清单 10-15

```

1  #include <iostream.h>
2  #include <math.h>                                //包含数学运算函数的头文件
3  class complex
4  {
5      double real, imag;
6  public:
7      complex(double real=0.0, double imag=0.0); //声明构造函数
8      ~complex();                               //声明析构函数
9      double abscomplex();
10 };
11 complex::complex(double r, double i)           //定义构造函数
12 {
13     cout<<"constructing....."<<endl;
14     real=r;                                     //成员初始化
15     imag=i;
16 }
17 complex::~~complex()                           //定义析构函数
18 {
19     cout<<"destructing....."<<endl;
20 }
21 double complex::abscomplex()                   //定义成员函数
22 {
23     double n;
24     n=real*real+imag*imag;
25     return sqrt(n);                             //返回平方根值
26 }
27 void main()
28 {
29     complex ob(1.1, 2.2);                       //创建对象调用构造函数
30     cout<<"abs of complex ob="<<ob.abscomplex()<<endl;
31 }                                                 //释放对象调用析构函数

```

【运行结果】在 Visual C++ 中执行上述代码，其结果如图 10-19 所示。

【范例解析】上述代码中，定义了一个类 `complex`，在类中声明了该类的构造函数和析构函数，在类外定义了构造函数和析构函数。在 `main()` 函数中创建对象 `ob`，调用构造函数，因此输出 “constructing.....” 信息。在输出 `ob` 对象的绝对值后，程序结束，此时编译系统自动调用了析构函数释放该对象，因此输出 “destructing.....” 信息。

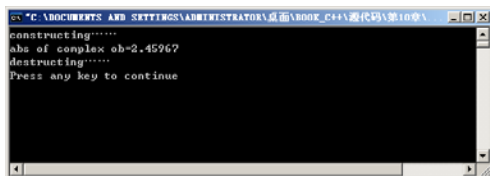


图 10-19 析构函数

在使用析构函数中，读者需要注意如下的几个问题：

- 每个类必须有一个析构函数，若没有显式地定义，则系统会自动生成一个默认的析构函数，它是一个空函数。
- 对于大多数类而言，默认的析构函数就能满足要求，但如果对象在完成操作前需要做内部处理，则应显式地定义析构函数。
- 构造函数和析构函数的常见用法是，在构造函数中用 `new` 运算符为对象分配空间，在析构函数中用 `delete` 运算符释放空间。



提示 上述程序中，由于使用到了计算平方根的函数 `sqrt()`，因此必须在头文件中调用 `math.h`，即添加语句 `#include <math.h>`。

10.6 友元

C++中，为了使得类的私有成员和保护成员能够被其他类或其他成员函数访问，引入了友元的概念。友元提供了不同类或对象的成员函数之间、类的成员函数与一般函数之间进行数据共享的机制。如果友元是一般成员函数或类的成员函数，则称为友元函数；如果友元是一个类，则称为友元类，友元类的所有成员函数都是友元函数。

10.6.1 友元函数

友元函数与普通成员函数不同，它不是当前类的成员函数，而是独立于当前类的外部函数；它可以是普通函数或其他类的成员函数。友元函数定义后可以访问该类的所有对象的成员，包括私有成员、保护成员和公有成员。

友元函数使用前必须要在类定义时声明，声明时在其函数名前加上关键字 `friend`。该声明可以放在公有成员中，也可以放在私有成员中。而友元函数的定义既可以在类内部进行，也可以在类外部进行，但通常都定义在类的外部。C++中，将普通函数声明为友元函数的一般形式为：

```
friend<数据类型><友元函数名>(参数表);
```

【范例 10-16】普通函数作为友元函数。该范例实现一个求两点之间距离的功能，该类中声明了一个友元函数 `dist()`，在外部对该友元函数进行了定义，实现代码如代码清单 10-16 所示。

代码清单 10-16

```
1  #include <iostream.h>
2  #include <math.h>                //调用头文件
3  class point                      //定义类
```




```

4  {
5      double x,y;                                //定义私有成员变量
6  public:
7      point(double a=0,double b=0)                //定义构造函数
8      {
9          x=a;                                    //初始化成员
10         y=b;
11     }
12     point(point &p);                             //重载构造函数
13     double getx()                                //定义成员函数
14     {
15         return x;
16     }
17     double gety()
18     {
19         return y;
20     }
21     friend double dist(point &p1,point &p2);      //声明友元函数
22 };
23 double dist(point &p1,point &p2)                //定义友元函数
24 {
25     return (sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y)));
26 }
27 void main()
28 {
29     point ob1(1,1);                              //创建对象
30     point ob2(4,5);
31     cout<<"The distance is:"<<dist(ob1,ob2)<<endl; //调用友元函数
32 }

```

【运行结果】在 Visual C++ 中执行上述代码，其结果如图 10-20 所示。

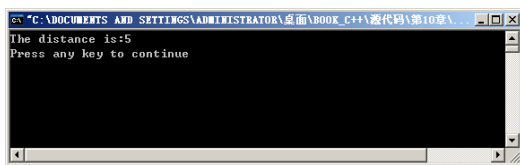


图 10-20 友元函数

【范例解析】上述代码定义了类 `point`，在类中声明了友元函数 `dist()`，在类的外部对该函数进行了定义。在主函数 `main()` 中创建了两个对象 `ob1` 和 `ob2`，创建对象的同时调用构造函数对其成员进行了初始化，因此，调用友元函数 `dist()` 时得到如图 10-20 所示的结果。

一般来说，使用友元函数应注意如下的问题：

- 由于友元函数不是成员函数，因此，在类外定义友元函数时，不必像成员函数那样，在函数名前加“类名::”。
- 友元函数不是类的成员，因而不能直接引用对象成员的名字，也不能通过 `this` 指针引用对象的成员，必须通过作为入口参数传递进来的对象名或对象指针来引用该对象的成员。为此，友元函数一般都带有一个该类的入口参数，如上例中的 `distance(point &p1,point &p2)`。
- 当一个函数需要访问多个类时，应该把这个函数同时定义为这些类的友元函数，这样，这个函数才能访问这些类的数据。



提示 读者可以看到，上述代码中的第 25 行友元函数的定义中，虽然其定义在类外，但由于它是友元函数，因此可以访问类的私有成员变量。

10.6.2 友元成员

如果一个类的成员函数是另一个类的友元函数，则称这个成员函数为友元成员。通过友元成员函数，不仅可以访问自己所在类对象中的私有和公有成员，还可访问由关键字 `friend` 声明语句所在的类对象中的私有和公有成员，从而可使两个类可以相互访问，从而共同完成某个任务。

【范例 10-17】友元成员的应用。该范例定义了友元成员，该成员函数实现输出两个类的输出，它可以同时访问两个类的私有和公有成员，其实现代码如代码清单 10-17 所示。

代码清单 10-17

```

1  #include <iostream.h>
2  #include <string.h>
3  class boy;                //友元函数prt()带了 girl 和 boy 两个类的对象
4                              //类 boy 要在后面才声明，提前声明它，以便使用该类的对象
5  class girl                //定义类 girl
6  {
7      char *name;           //定义私有成员
8      int age;
9  public:
10     girl(char *n,int a)    //定义构造函数
11     {
12         name=new char[strlen(n)+1]; //分配空间
13         strcpy(name,n);      //调用字符串拷贝函数
14         age=a;
15     }
16     void prt(boy &);        //声明公有成员函数
17     ~girl()                //定义析构函数
18     {
19         delete name;        //释放空间
20     }
21 };
22 class boy                  //定义类 boy
23 {
24     char *name;
25     int age;
26 public:
27     boy(char *n,int a)      //定义构造函数
28     {
29         name=new char[strlen(n)+1];
30         strcpy(name,n);
31         age=a;
32     }
33     friend void girl::prt(boy &); //声明友元成员
34     ~boy()                  //定义析构函数
35     {
36         delete name;        //释放空间
37     }
38 };
39 void girl::prt (boy &b)     //定义友元成员
40 {
41     cout<<"girl\'s name:"<<name<<"    age:"<<age<<"\n";
42     cout<<"boy\'s name:"<<b.name<<"    age:"<<b.age<<"\n";
43 }
44 void main()
45 {
46     girl g("Stacy",15);    //创建类 girl 的对象
47     boy bl("Jim",16);      //创建类 boy 的对象
48     g.prt(bl);

```



49 }

【运行结果】在 Visual C++ 中执行上述代码，其结果如图 10-21 所示。

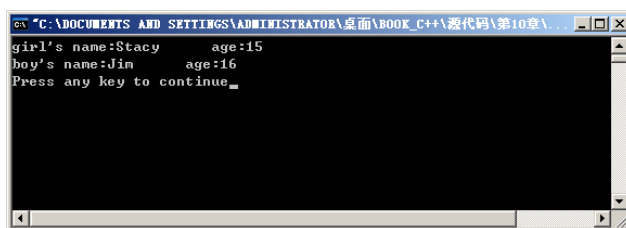


图 10-21 友元成员

【范例解析】上述代码中定义了 boy 和 girl 两个类，在类 boy 中声明了 girl 类的 prt() 函数为其友元成员，在类外定义了该成员函数。读者可以看到，该函数既可访问 girl 类的成员变量 age，又能够访问 boy 类的成员 age。

需要读者注意的是，当一个类的成员函数作为另一个类的友元函数时，必须先定义成员函数所在的类，如范例 10-17 中，类 girl 的成员函数 prt() 为类 boy 的友元函数，就必须先定义类 girl。并且在声明友元函数时，要加上成员函数所在类的类名和运算符 “::”，如范例 10-17 中的语句：

```
friend void girl::prt(boy &);
```

另外，在主函数中一定要创建一个类 girl 的对象。只有这样，才能通过对象名调用友元函数。如范例 10-17 中主函数的语句：

```
girl g("Stacy",15);
```



如果在类定义前要使用到该类的成员，需要事先在使用前对该类进行声明，如上述代码中的声明语句 “class boy;”，否则系统将报错。

10.6.3 友元类

当一个类作为另一个类的友元时，称这个类为友元类。当一个类成为另一个类的友元类时，这个类的所有成员函数都成为另一个类的友元函数，因此，友元类中的所有成员函数都可以通过对象名直接访问另一个类中的私有成员，从而实现了不同类之间的数据共享。

C++ 中，友元类的声明可以放在类声明中的任何位置，这时，友元类中的所有成员函数都称为友元函数。友元类声明的一般形式如下：

```
friend class <友元类名>; 或 friend <友元类名>;
```

【范例 10-18】友元类的应用。该范例实现与范例 10-17 相同的功能，创建对象后调用函数将其初始化的结果输出，使用友元类来实现，代码如代码清单 10-18 所示。

代码清单 10-18

```
1  #include <iostream.h>
2  #include <string.h>
3  class boy;
4                                     //友元函数prt()带了girl和boy两个类的对象
                                     //类boy要在后面才声明，所以提前声明它，以便
                                     //使用该类的对象
5  class girl
6  {
7      char *name;                    //私有成员
```

```

8     int age;
9     public:
10    girl(char *n,int a)           //定义构造函数
11    {
12        name=new char[strlen(n)+1]; //分配空间
13        strcpy(name,n);
14        age=a;
15    }
16    void prt(boy &);              //声明公有成员函数
17    ~girl()                       //定义析构函数
18    {
19        delete name;              //释放空间
20    }
21 };
22 class boy                        //定义类 boy
23 {
24     char *name;                  //定义私有成员
25     int age;
26     friend girl;                 //声明类 girl 为类 boy 的友元类
27     public:
28     boy(char *n,int a)           //定义构造函数
29     {
30         name=new char[strlen(n)+1];
31         strcpy(name,n);
32         age=a;
33     }
34     friend void girl::prt(boy &); //声明友元成员
35     ~boy()                       //定义析构函数
36     {
37         delete name;
38     }
39     void girl::prt (boy &b)       //定义友元成员
40     {
41         cout<<"girl\'s name:"<<name<<"    age:"<<age<<"\n";
42         cout<<"boy\'s name:"<<b.name<<"    age:"<<b.age<<"\n";
43     }
44     void main()
45     {
46         girl g("Stacy",15);      //创建类 girl 的对象
47         boy bl("Jim",16);         //创建类 boy 的对象
48         g.prt(bl);
49     }

```

【运行结果】在 Visual C++中执行上述代码，其结果如图 10-22 所示。

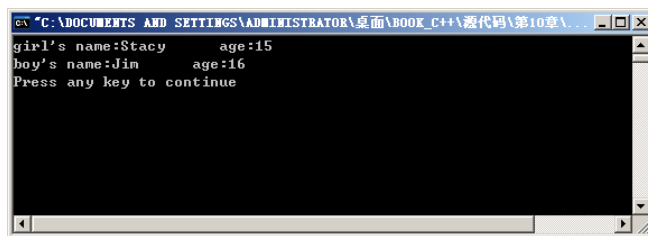


图 10-22 友元类

【范例解析】上述代码同样定义了 boy 类和 girl 类，与代码 10-17 不同的是，此处，在 boy 类的定义中将 girl 类声明为 boy 类的友元类，因此，girl 类的成员可以访问 boy 类的任意成员和函数。那么，在函数 prt()中，也可以直接调用 boy 类的私有成员变量 age 并输出。


注意

友元关系是不能传递的。类 B 是类 A 的友元，类 C 是类 B 的友元，类 C 与类 A 之间，除非特别声明，没有任何关系，不能进行数据共享。友元关系是单向的。类 B 是类 A 的友元，类 B 的成员函数可以访问类 A 的私有成员和保护成员，反之，类 A 的成员函数却不可以访问类 B 的私有成员和保护成员。

10.7 小结

本章主要介绍了 C++ 的面向对象程序设计特点之一的类和对象，类和对象是所有面向对象程序设计语言的基本特征。本章开始介绍了类的概念、引入，以及类的声明和定义，着重讲解了类的成员的访问控制，接着介绍了对象的概念和对象数组的使用。作为重点和难点，本章重点介绍了类的构造函数和析构函数的定义和使用。其中，对类的构造函数做了详细讲解，介绍了构造函数的定义、调用，以及重载构造函数和拷贝构造函数的定义和使用。最后，本章通过综合练习让读者回顾了类和对象的定义使用。

10.8 习题

1. 分析以下程序执行的结果，并写出其输出结果。

```
#include<iostream.h>
class Sample
{
public:
    int x,y;
    Sample(){x=y=0;}
    Sample(int a,int b){x=a;y=b;}
    void disp()
    {
        cout<<"x="<<x<<" ,y="<<y<<endl;
    }
};
void main()
{
    Sample s1(2,3);
    s1.disp();
}
```

【解答】该习题主要考查重载构造函数的定义方法。读者需要注意，构造函数是唯一不能被显式调用的成员函数，它在定义类的对象时自动调用，也称为隐式调用。该习题首先定义了一个类 Sample，在 main() 中定义了它的一个对象，定义 s1 对象时调用其重载构造函数(x=2,y=3)，然后，调用其成员函数输出数据成员。因此，该程序段输出为：x=2，y=3。

2. 编写一个程序，输入 N 个学生数据，包括学号、姓名、成绩，要求输出这些学生数据并计算平均分。

【解答】该习题主要考查类的定义。此处设计一个学生类 Stud，除了包括 no(学号)、name(姓名)和 deg(成绩)数据成员外，有两个静态变量 sum 和 num，分别存放总分和人数，另有两个普通成员函数 setdata() 和 disp()，分别用于给数据成员赋值和输出数据成员的值，另有一个静态成员函数 avg()，用于计算平均分。在 main() 函数中定义了一个对象数组用于存储输入的学生数据。其简要的实现代码如下所示。

```
class Stud
{
    int no;
```

```

char name[10];
int deg;
static int num;
static int sum;
public:
    void setdata(int n,char na[],int d)
    {
        no=n; deg=d;
        strcpy(name,na);
        sum+=d;
        num++;
    }
    static double avg()
    {
        return sum/num;
    }
    void disp()
    {
        cout<<no<<name<<deg;
    }
};

```

3. 分析以下程序的执行结果。

```

#include<iostream.h>
class Sample
{
public:
    int x;
    int y;
    void disp()
    {
        cout<<"x="<<x<<" ,y="<<y<<endl;
    }
};
void main()
{
    int Sample::*pc;
    Sample s;
    pc=&Sample::x;
    s.*pc=10;
    pc=&Sample::y;
    s.*pc=20;
    s.disp();
}

```

【解答】该习题主要考查类数据成员指针的使用方法。在 `main()` 中定义的 `pc` 是一个指向 `Sample` 类数据成员的指针。执行 `pc=&Sample::x` 时, `pc` 指向数据成员 `x`, 语句 `s.*pc=10` 等价于 `s.x=10` (为了保证该语句正确执行, `Sample` 类中的 `x` 必须是公共成员); 执行 `pc=&Sample::y` 时, `pc` 指向数据成员 `y`, 语句 `s.*pc=20` 等价于 `s.y=20` (同样, `Sample` 类中的 `y` 必须是公共成员)。所以输出为: `x=10,y=20`。

4. 分析以下程序的执行结果。

```

#include<iostream.h>
class Sample
{
    int x,y;
public:
    Sample(){x=y=0;}
    Sample(int a,int b){x=a;y=b;}
    void disp()
    {

```



```

        cout<<"x="<<x<<" ,y="<<y<<endl;
    }
};
int main()
{
    Sample s1,s2(2,3);
    s1.disp();
    s2.disp();
}

```

【解答】该习题主要考查构造函数的调用顺序。上述程序段首先定义了一个类 Sample，在 main() 中定义了它的两个对象，定义 s1 对象时调用其默认构造函数(x=0,y=0)，定义 s2 对象时调用其重载构造函数(x=2,y=3)，然后，调用各自的成员函数输出各自的数据成员。所以输出为：

```

x=0,y=0
x=2,y=3

```

5. 定义一个类 CPoint，在该类中定义构造函数、重载构造函数和析构函数，在 main() 函数中调用该类创建对象 pt，查看该对象调用成员函数的输出情况，如图 10-23 所示。

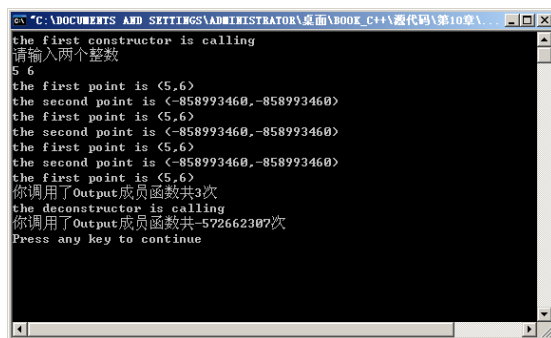


图 10-23 类和对象的使用

【解答】该程序段可通过调用 CPoint pt; 语句来产生一个对象时，同时相应地会去调用构造函数 CPoint()，而以带参数的定义对象语句形式来产生一个对象时，就会调用带参数的构造函数。其简要的实现代码如下所示。

```

class CPoint                                     //定义类 CPoint
{
public:                                           //定义公有成员
    int x1;                                     //公有数据成员
    int y1;
    void Output();                             //成员函数
    CPoint();                                   //构造函数
    CPoint(int x2,int y2);                     //重载构造函数
    ~CPoint();                                 //析构函数
private:                                        //定义私有成员
    int x2;                                    //私有数据成员
    int y2;
    int *pCount;                               //指针变量
};
void CPoint::Output()                           //定义成员函数
{
    if(pCount)                                 //pCount 指针不为空
        (*pCount)++;                          //该指针指向的值递增 1
    else
    {
        pCount=new int;                       //申请整型空间
    }
}

```

```

        *pCount=1;                                //指针变量初始化
    }
    cout<<"the first point is ("<<x1<<','<<y1<<')"<<endl;
                                           //输出
    cout<<"the second point is ("<<x2<<','<<y2<<')"<<endl;
}
CPoint::CPoint()                                //定义构造函数
{
    pCount=0;                                    //指针初始化
    cout<<"the first constructor is calling"<<endl; //输出信息
}
CPoint::CPoint(int x2,int y2)                    //重载构造函数
{
    this->x2=x2;                                  //成员变量赋值
    this->y2=y2;
    pCount=0;
    cout<<"the second constructor is calling"<<endl; //输出信息
}
CPoint::~~CPoint()                               //定义析构函数
{
    if(pCount)                                    // pCount 指针不为空
    {
        cout<<"你调用了 Output 成员函数共"<<*pCount<<"次"<<endl;
        delete pCount;                            //释放空间
    }
    else                                           // pCount 为空
        cout<<"你还没有调用过 Output 成员函数"<<endl; //输出信息
    cout<<"the deconstructor is calling"<<endl;
}
void Output(CPoint pt)                           //定义成员函数
{
    cout<<"the first point is ("<<pt.x1<<','<<pt.y1<<')"<<endl;
                                           //输出结果
    //cout<<"the second point is ("<<pt.x2<<','<<pt.y2<<')"<<endl;
    //上面被注释的语句会造成编译错误, 因为不能从类的外部访问类中的私有成员
}
void main()                                       //主函数
{
    if(1=1)                                       //限定 pt 变量的有效范围
    {
        CPoint pt;                               //创建对象
        cout<<"请输入两个整数"<<endl;            //输入提示
        cin>>pt.x1>>pt.y1;                       //接收键盘输入
        //pt.x2=10;
        //pt.y2=10;
        //上面被注释的语句会造成编译错误, 因为不能从类的外部访问类中的私有成员
        pt.Output();                               //调用对象的成员函数
        pt.Output();
        pt.Output();                               //故意演示 Output 被调用多次的情况
        Output(pt);
    }
    CPoint pt(10,10);                             //创建对象
    pt.Output();                                   //调用对象成员函数
}

```


第 11 章 继承

继承是面向对象程序设计的重要特性之一。C++作为一种面向对象程序设计语言，提供了丰富的继承功能。类的继承是新的类从已有类那里得到已有的特性，从已有的类产生新类的过程就是类的派生。在继承过程中，原有的类或已经存在的用来派生新类的类称为基类或父类，而由已经存在的类派生出的新类则称为派生类或子类。本章主要介绍类的继承方式、派生类、多重继承，以及多重继承之间的访问控制机制。

以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为一个学习的参照。

- 了解 C++ 中继承与派生的概念。
- 掌握 C++ 支持的派生方式及派生类的构造函数和析构函数的定义和使用。
- 掌握多重继承和虚基类的应用。

11.1 继承与派生

继承是面向对象的一块基石，其允许创建分等级层次的类。利用继承可以创建一个通用的类，然后由更具体的类来继承它，再在这些类里加入自己新的成员。

11.1.1 继承与派生概述

前面介绍面向对象程序设计时提到，封装、继承和多态是面向对象程序设计方法的三个特征。而继承是指一个类除了得到另外一个类的所有性质外，还具有自身独特的性质。则该类称为派生类，而另外一个类称为基类，这种行为称为继承。

例如，在现实生活中，人是一个类，动物也是一个类，而人这个类具有动物这个类的所有特征，此外，还具有其他动物所没有的独特性质，如说话、直立行走等。因此，在该关系中，人是派生类，动物是基类，人这个类继承于动物。

继承是面向对象的一个重要特征，根据派生类所拥有的基类数目不同，可以分为单继承和多继承。一个类只有一个直接基类时，称为单继承；而一个类同时有多个直接基类时，则称为多继承，如图 11-1 所示。

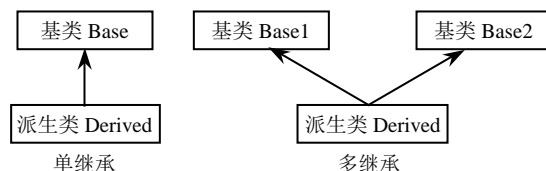


图 11-1 继承的种类

基类与派生类之间的关系如下：

- 基类是对派生类的抽象，派生类是对基类的具体化，是基类定义的延续。
- 派生类是基类的组合，多继承可以看做是多个单继承的简单组合。
- 公有派生类的对象可以作为基类的对象处理。

11.1.2 声明派生类

通过前面的介绍,读者已经知道了,对于已有的类称为基类,由基类继承而来的类称为派生类。而派生类定义的一般形式是:

```
class <派生类名>:<派生方式><基类名>
{
    派生类成员声明;
}
```

其中,参数说明如下:

- 继承方式关键字为 `private`、`public` 和 `protected`,分别表示私有继承、公有继承和保护继承。默认的继承方式是私有继承。继承方式规定了派生类成员和类外对象访问基类成员的权限,有关内容将在后面章节介绍。
- 派生类成员是指除了从基类继承来的成员外,新增加的数据成员和成员函数。正是通过在派生类中新增加成员来添加新的属性和功能,来实现代码的复用和功能的扩充。



注意 C++中,如果没有指定派生方式,即为 `private` 私有继承,也就是说, `private` 继承方式为 C++ 的默认派生方式,这是为了保护数据的安全性。

例如,下列语句声明了两个类 A 和 B,其中类 A 作为基类,类 B 公有继承类 A,即类 B 为类 A 的派生类。

```
class A
{
private:
    int s;
public:
    void inits(int n);
};
class B:public A //类 A 以公有继承的方式派生类 B
{
private:
    int t;
public:
    void initt(int n)
};
```

采用上述语句声明派生类 B 后,由类 B 创建的对象即可以访问类 B 的成员,也可以对类 A 的成员进行访问,但访问的权限由继承方式来决定。

11.2 访问控制

前面内容提到了,类的继承方式有公有继承(`public`)、保护继承(`protected`)和私有继承(`private`)三种,不同的继承方式,导致原来具有不同访问属性的基类成员在派生类中的访问属性也有所不同。

总的来说,派生类对基类成员的访问能力如表 11-1 所示。



提示 表 11-1 中, `private/public/protect` 分别表示在派生类中为私有成员、公有成员和保护成员,不可访问表示派生类中不可以访问基类的私有成员。



表 11-1 派生类对基类成员的访问能力

基类成员 \ 继承方式	公有继承 (public)	私有继承 (private)	保护继承 (protected)
私有成员 private	private	不可访问	不可访问
公有成员 public	public	private	protected
保护成员 protected	protected	private	protected

从表 11-1 中读者可以看出，关于派生类访问基类的能力主要可以描述如下：

- 基类中的私有成员在派生类中是隐藏的，只能在基类内部访问。
- 派生类中的成员不能访问基类中的私有成员，但可以访问基类中的公有成员和保护成员。
- 派生类从基类公有继承时，基类的公有成员和保护成员在派生类中仍为公有成员和保护成员。
- 派生类从基类私有继承时，基类的公有成员和保护成员在派生类中都改变为私有成员。
- 派生类从基类保护继承时，基类的公有成员在派生类中改变为保护成员，基类的保护成员在派生类中则仍为保护成员。

11.2.1 公有继承

在公有继承中，基类成员的可访问性在派生类中保持不变，即基类的私有成员在派生类中还是私有成员，不允许外部函数和派生类的成员函数直接访问，但可以通过基类的公有成员函数访问。基类的公有成员和保护成员在派生类中仍是公有成员和保护成员，派生类的成员函数可直接访问它们，而外部函数只能通过派生类的对象间接访问它们。

【范例 11-1】 公有继承的应用。该范例定义了类 A 作为基类，定义类 B 公有继承于类 A，读者观察其成员的属性，代码如代码清单 11-1 所示。

代码清单 11-1

```

1  #include <iostream.h>
2  class A                                //定义基类 A
3  {
4  private:                                //定义基类私有成员
5      int s;
6  public:                                 //定义基类公有成员
7      void inits(int n)                  //定义成员函数
8      {
9          s=n;
10     }
11     int gets()                          //定义成员函数
12     {
13         return s;
14     }
15 };
16 class B:public A                        //类 A 以公有继承的方式派生类 B
17 {
18 private:                                //定义派生类的私有成员变量
19     int t;
20 public:
21     void initt(int n)                   //定义派生类的成员函数
22     {
23         t=n;
24     }
25     int gett()                          //定义派生类的成员函数

```

```

26     {
27         return t*gets();           //调用基类成员函数
28     }
29 };
30 void main()
31 {
32     B ob;                          //创建对象
33     ob.inits(12);                  //通过类外的对象访问基类的公有成员
34     ob.initt(5);                  //调用派生类的公有成员
35     cout<<"the result of ob.gett() is: "<<ob.gett()<<endl;
36 }

```

【运行结果】在 Visual C++ 中新建一个【C++ Source File】，在其中输入如上的代码，编译无误后运行，其结果如图 11-2 所示。

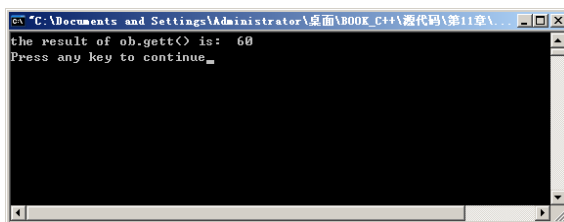


图 11-2 公有继承

【范例解析】上述代码中，定义了两个类 A 和 B，其中类 B 公有继承于类 A。在主函数 main() 中，由类 B 创建了一个对象 ob，通过该对象可以调用类 A 的公有成员函数 inits()，也可以访问其自身类 B 的成员函数。此处注意如下的两个问题：

- 虽然派生类以公有的方式继承了基类，但并不是说派生类就可以访问基类的私有成员，基类无论怎样被继承，其私有成员对派生类而言仍然保持私有性。
- 在派生类中声明的名字如果与基类中声明的名字相同，则派生类中的名字起支配作用。也就是说，若在派生类的成员函数中直接使用该名字的话，该名字是指在派生类中声明的名字。



注意 根据如上的说法，如果要使用基类中的名字，则应使用作用域运算符加以限定，即在该名字前加“基类名::”。例如：

```

class base
{
public:
    int f();
};
class derived:public base
{
    int f();
    int g();
};
void derived::g()
{
    f();           //被调用的函数是 derived:: f()而不是 base:: f()
}

```

上述结论，也适用于派生类的对象的引用，例如：

```

derived obj;
obj.f();           //被调用的函数是 derived:: f()而不是 base:: f()

```

要使用基类中的名字，则应用作用域运算符限定，例如：

```

obj. base::f();

```



11.2.2 私有派生

在私有派生中，派生类只能以私有方式继承基类的公有成员和保护成员，因此，基类的公有成员和保护成员在派生类中成为私有成员，它们能被派生类的成员函数直接访问，但不能被类外函数访问，也不能在类外通过派生类的对象访问。另外，基类的私有成员派生类仍不能被访问。



提示 在设计基类时，通常都要为其私有成员提供公有的成员函数，以便派生类和外部函数能间接访问它们。

【范例 11-2】私有派生的应用。该范例定义了类 A 作为基类，定义类 B 私有继承于类 A，请读者观察其成员的属性，代码如代码清单 11-2 所示。

代码清单 11-2

```

1  #include <iostream.h>
2  class A                                //定义基类 A
3  {
4  private:                                //定义基类私有成员
5      int s;
6  public:                                  //定义基类公有成员
7      void inits(int n)                  //定义成员函数
8      {
9          s=n;
10     }
11     int gets()                          //定义成员函数
12     {
13         return s;
14     }
15 };
16 class B:private A                      //类 A 以私有继承的方式派生类 B
17 {
18 private:                                //定义派生类的私有成员变量
19     int t;
20 public:
21     void init(int n)                    //定义派生类的成员函数
22     {
23         t=n;
24     }
25     int get()                          //定义派生类的成员函数
26     {
27         return t*gets();                //调用基类成员函数
28     }
29 };
30 void main()
31 {
32     B ob;                               //创建派生类对象
33     ob.inits(12);                        //非法，不能通过类外对象访问从基类私有继承来的成员
34     ob.initst(5,7);
35     cout<<ob.get()<<endl;              //调用派生类成员函数
36 }
```

【运行结果】在 Visual C++ 中输入如上的代码，编译后，其返回如图 11-3 所示的错误信息。

【范例解析】上述代码中，错误来源于第 35 行，该行代码通过类 B 创建的对象对类 A 中的公有成员进行调用，但由于类 B 是私有继承于类 A，因此类 A 的公有成员在类 B 中被称为私有成员，从而不能被外部函数所访问，因此，上述代码运行出错。



图 11-3 错误信息

要成功运行上述程序，只需将上述代码的第 35 行注释掉，使其不能访问基类的成员即可，读者可自行查看其运行结果。

提示 在实际应用中，由于基类经过多次派生以后，其私有成员可能会成为不可访问的，所以用得比较少。

11.2.3 保护继承

通过前面 11.2.1 和 11.2.2 节的学习，读者可以知道，不论是公有派生还是私有派生，派生类都不能访问它的基类的私有成员，要想访问，只能通过调用基类成员函数的方式来实现，也就是使用基类提供的接口来访问。对于需要频繁访问基类私有成员的派生类而言，这种方式较为不便。

为了解决这个问题，C++ 提供了具有另一种访问特性的成员——保护（protected）成员。保护成员可被本类或派生类的成员函数访问，但不能被外部函数访问。为便于派生类的访问，可将基类中的需要提供给派生类访问的私有成员定义为保护成员。

1. 保护成员的声明

保护成员用关键字 `protected` 声明，它可以放在类声明的任何地方，通常放在私有成员和公有成员之间，其声明的一般形式为：

```
class 类名
{
    private:
        //私有成员
    protected:
        //保护成员
    public:
        //公有成员
};
```

【范例 11-3】声明保护成员。该范例声明一个类 A，其包含保护成员变量，在主函数中对该保护成员进行访问，读者可理解其访问控制机制，代码如代码清单 11-3 所示。

代码清单 11-3

```
1  #include<iostream.h>
2  class A                                //定义基类 A
3  {
4  private:                                //定义私有成员
5      int s;
6  protected:
7      int r;                                //声明变量 r 为保护成员
8  public:                                  //定义公有成员
9      int t;
10     void setsr(int n,int m)             //定义基类成员函数
11     {
12         s=n;
13         r=m;
```



```

14     }
15     int gets()                //定义基类成员函数
16     {
17         return s;
18     }
19     int getr()                //定义基类成员函数
20     {
21         return r;
22     }
23 };
24 void main()
25 {
26     A ob;
27     //ob.s=10;                //非法，外部函数不能访问类的保护成员
28     //ob.r=20;                //非法，外部函数不能访问类的保护成员
29     ob.t=30;                  //合法，外部函数能访问类的公有成员
30     cout<<ob.gets()<<"      "<<ob.getr()<<"      "<<ob.t<<endl;
31     ob.setsr(10,20);          //通过调用公有成员函数给私有成员变量赋值
32     cout<<ob.gets()<<"      "<<ob.getr()<<"      "<<ob.t<<endl;
33 }

```

【运行结果】在 Visual C++中执行上述代码，其运行结果如图 11-4 所示。

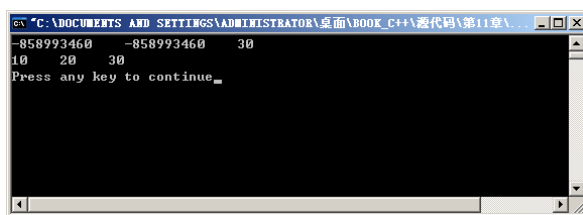


图 11-4 保护成员的访问

【范例解析】上述代码中定义类 A 中包含了保护成员变量 r，而在主函数 main()中，由类 A 创建了对象 ob，该对象直接访问私有成员 s 和保护成员 r 都是非法的，因此将代码注释掉了，对象 ob 只能直接访问公有变量 t。



注意 如果要访问私有成员和保护成员，必须通过公有成员函数来访问，上述范例中是通过 setsr()函数来实现的。

了解了保护成员的概念后，再来看保护成员被继承后访问特性的变化。由于继承的方式不同，因此保护成员被继承后其访问特性相应如下：

- 若为公有派生，则基类中的保护成员在派生类中也为保护成员，可被派生类直接访问，但不能被外部函数直接访问，外部函数只能通过派生类的对象间接访问它们。
- 若为私有派生，则基类中的保护成员在派生类中成为私有成员，派生类可直接访问它们，但外部函数或在类外通过派生类的对象都不能访问它们。

2. 保护继承

在保护继承中，基类的公有成员在派生类中成为保护成员，基类的保护成员在派生类中仍为保护成员，所以，派生类的所有成员在类的外部都无法访问它们。

【范例 11-4】保护继承的应用。该范例定义了类 A 作为基类，定义类 B 保护继承于类 A，读者观察其成员的属性，代码如代码清单 11-4 所示。

代码清单 11-4

```

1  #include<iostream.h>
2  class A                                //定义基类 A
3  {
4  private:                                //定义基类私有成员
5      int s;
6  protected:
7      int r;                              //声明变量 r 为保护成员
8  public:
9      int t;
10     void setst(int n,int m)             //定义基类公有成员函数
11     {
12         s=n;
13         t=m;
14     }
15     int gets()                          //定义基类公有成员函数
16     {
17         return s;
18     }
19 };
20 class B:protected A                    //类 B 以保护继承的方式继承类 A 的成员
21 {
22 private:                                //定义派生类私有成员
23     int p;
24 public:
25     void setsrp(int n,int m,int l) //定义派生类公有成员函数
26     {
27         setst(n,m);
28         r=m;                            //可直接访问从基类中保护继承的成员
29         p=l;
30     }
31     int getp()
32     {
33         p=p+t*gets();                   //通过基类的成员函数间接访问基类的私有成员
34         return p;
35     }
36 };
37 void main()
38 {
39     B ob;
40     // ob.setst(12,12);                 //非法, 不能通过类外对象访问从基类保护继承来的成员
41     ob.setsrp(12,12,5);                 //合法, 访问派生类本身的成员函数
42     cout<<"the result of ob.getp() is: "<<ob.getp()<<endl;
43 }

```

【运行结果】在 Visual C++中执行上述代码, 其运行结果如图 11-5 所示。

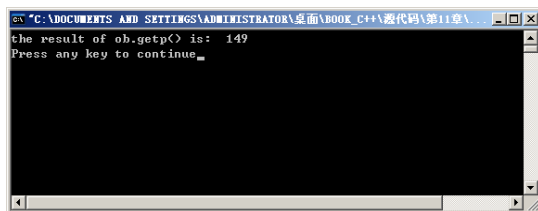


图 11-5 保护继承

【范例解析】上述代码中, 类 A 中定义了保护成员变量 r, 类 B 保护继承于类 A。在主函数 main()中, 由类 B 创建对象 ob, 该对象直接访问类 A 的公有成员函数 setst()时出现错误, 这



是因为经过保护继承后，类 A 的公有成员在类 B 中成为保护成员，而外部函数是不能直接访问保护成员的，需要通过调用公有成员函数来实现。



提示 当对上述代码的第 40 行加上注释符号“//”后，表示不编译该语句，整个程序就可以被顺利执行，否则将出现编译错误。

3. 派生类直接访问基类成员

前面内容提到了，派生类不能直接访问基类的私有成员，若要访问，必须使用基类的接口，即通过其成员函数。实现的方法有如下两种：

- 在基类的声明中增加保护成员，将基类中提供给派生类访问的私有成员定义为保护成员。
- 将需要访问基类私有成员的派生类成员函数声明为友元。

【范例 11-5】派生类直接访问基类成员。该范例定义的派生类直接访问基类 A 中的成员，读者观察哪些可以访问，哪些不可以访问，代码如代码清单 11-5 所示。

代码清单 11-5

```

1  #include <iostream.h>
2  class A                                //定义基类 A
3  {
4      friend class C;                    //声明类 C 为友元
5      int x;                             //基类的私有成员
6  protected:
7      int y;                             //基类的保护成员
8  };
9  class B:A                              //类 A 私有派生类 B
10 {
11 public:
12     // int getx()                       //非法，不能直接访问基类的私有成员
13     // {
14     //     return x;
15     // }
16     int gety()                           //正确，能直接访问基类的保护成员
17     {
18         return y;
19     }
20 };
21 class C:A                               //类 C 私有继承于类 A
22 {
23 public:                                  //声明派生类 C 的公有成员函数
24     int getx();
25 };
26 int C::getx()                           //定义派生类 C 的公有成员函数
27 {
28     C ob;
29     return ob.x;                         //正确，友元能直接访问基类的私有成员
30 }
31 void main()
32 {
33     int i;
34     C ob;                                //创建对象
35     i=ob.getx();                         //调用成员函数
36     cout<<i<<endl;
37 }
```

【运行结果】在 Visual C++ 中执行上述代码，其运行结果如图 11-6 所示。

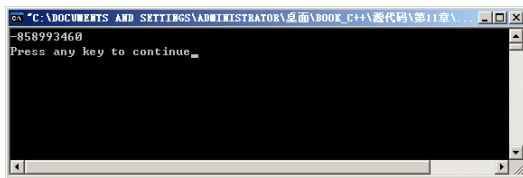


图 11-6 派生类直接访问基类成员

【范例解析】上述代码中，定义了三个类 A、B 和 C，其中类 B 和类 C 都是私有继承于类 A，在类 B 的定义中，成员函数 getx() 直接访问基类 A 的私有成员 x，这是错误的。而在类 C 的定义中，成员函数 getx() 直接访问基类 A 的私有成员 x，这是正确的，由于在类 A 的定义中声明了类 C 是类 A 的友元，因此类 C 可以直接访问类 A 的私有成员。

但是，由于上述范例没有指明 x 的值，因此在类 C 中访问基类 A 的私有成员 x 的值时，其值为一个无意义的数字。



注意 在类的继承中，如果没有指明继承方式，Visual C++ 默认为私有继承。即上述语句中的“class B:A”相当于“class B:private A”。

11.3 派生类的构造函数和析构函数

C++ 规定，基类成员的初始化工作由基类的构造函数完成，而派生类的初始化工作由派生类的构造函数完成，因此读者需要对派生类的构造函数和析构函数有一些了解。

11.3.1 执行顺序和构建原则

由于基类和派生类都需要调用构造函数来实现初始化成员，这就产生了派生类构造函数和析构函数的执行顺序问题，即当创建一个派生类的对象时，如何调用基类和派生类的构造函数分别完成各自成员的初始化，当撤销派生类对象时，又如何调用基类和派生类的析构函数分别完成各自的善后处理。

在构建派生类的构造函数和析构函数时，需要注意以下的构建原则：

- 基类的构造函数和析构函数不能被派生类继承。
- 如果基类没有定义构造函数，派生类也可以不定义构造函数，全都采用默认的构造函数，此时，派生类新增成员的初始化工作可用其他公有函数来完成。
- 如果基类定义了带有形参表的构造函数，派生类就必须定义新的构造函数，提供一个将参数传递给基类构造函数的途径，以便保证在基类进行初始化时能获得必需的数据。
- 如果派生类的基类也是派生类，则每个派生类只需负责其直接基类的构造，不负责自己的间接基类的构造。
- 派生类是否要定义析构函数与所属的基类无关，如果派生类对象在撤销时需要做清理善后工作，就需要定义新的析构函数。

11.3.2 派生类的构造函数

派生类的数据成员由所有基类的数据成员和派生类新增的数据成员共同组成，如果派生类新增成员中还有对象成员，那派生类的数据成员中还间接含有这些对象的数据成员。因此，要对派生类对象初始化，就要对基类数据成员、新增数据成员和对象成员的数据进行初始化。这



样, 派生类的构造函数需要以合适的初值作为参数, 隐含调用基类的构造函数和新增对象成员的构造函数来初始化各自的数据成员, 再用新加的语句对新增数据成员进行初始化。派生类构造函数声明的一般形式为:

```
<派生类名>::<派生类名>(参数总表):基类名(参数表),对象成员名1(参数表1),...,对象成员名n(参数表n)
{
    //派生类新增成员的初始化语句
}
```

其中, 参数的注意事项如下:

- 派生类的构造函数名与派生类名相同。
- 参数总表列出初始化基类成员数据、新增对象成员数据和派生类新增成员数据所需要的全部参数。
- 冒号后列出需要使用参数进行初始化的基类的名字和所有对象成员的名字及各自的参数表, 之间用逗号分开。对于使用默认构造函数的基类或对象成员, 可以不给出类名或对象名及参数表。

例如, 下列语句定义了两个类, 其中类 B 公有继承于类 A, 读者仔细观察其基类 A 和派生类 B 的构造函数的不同。

```
class A
{
    int i;
public:
    A(int n);                //声明基类的构造函数
}
class B:public A             //公有方式派生新类
{
    int j;                  //派生类新增数据成员
    A obj;                  //基类对象作为派生类对象成员
public:
    B(int n):A(n),ob(n);    //声明派生类的构造函数
}
```



注意 上述语句中声明了构造函数 B(int n), 由于是在类的内部进行声明的, 因此没有在其前加上派生类名和域运算符“::”, 如果在外部定义则需要加上。

11.3.3 派生类析构函数的构建

派生类析构函数的功能与基类析构函数的功能一样, 也是在对象撤销时进行必需的清理善后工作。析构函数不能被继承, 如果需要, 则要在派生类中重新定义。跟基类的析构函数一样, 派生类的析构函数也没有数据类型和参数。

派生类析构函数定义的方法与基类的析构函数的定义方法完全相同, 而函数体只需完成对新增成员的清理和善后工作就行了, 基类和对象成员的清理和善后工作, 编译系统会自动调用其各自的析构函数来完成。

【范例 11-6】派生类构造函数和析构函数的应用。该范例输出了基类和派生类构造函数和析构函数执行的顺序, 其实现代码如代码清单 11-6 所示。

代码清单 11-6

```
1  #include<iostream.h>
2  class base                //定义基类
3  {
```

```

4         int i;                                //定义基类私有成员
5     public:
6         base(int n)                            //基类构造函数
7         {
8             cout<<"constructing base class\n";
9             i=n;
10        }
11        ~base()                                //基类析构函数
12        {
13            cout<<"destructing base class\n";
14        }
15        void showi()                            //定义基类成员函数
16        {
17            cout<<i<<endl;
18        }
19    };
20    class derived:public base                    //公有方式派生新类
21    {
22        int j;                                //派生类新增数据成员
23        base obj;                            //基类对象作为派生类对象成员
24    public:
25        derived(int n):base(n),obj(n)          //派生类构造函数
26        {
27            cout<<"constrycting derived class"<<endl;
28            j=n;
29        }
30        ~derived()                            //派生类析构函数
31        {
32            cout<<"destructing derived class\n";
33        }
34        void showj()                            //定义派生类成员函数
35        {
36            cout<<j<<endl;
37        }
38    };
39    void main()
40    {
41        derived ob(10);                        //创建派生类对象
42        ob.showi();                            //调用基类成员函数
43        ob.showj();                            //调用派生类成员函数
44    }

```

【运行结果】在 Visual C++中执行上述代码，其运行结果如图 11-7 所示。

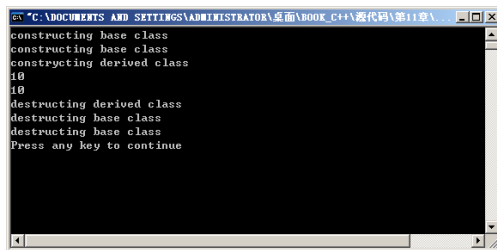


图 11-7 派生类的构造函数和析构函数

【范例解析】上述代码中定义了两个类 base 和 derived, 其中, 类 derived 公有继承于类 base。在两个类中都定义了构造函数和析构函数, 在主函数 main()中创建对象时读者可以看出其构造函数的执行顺序: 先调用基类的构造函数, 后调用派生类的构造函数。在删除对象时, 其析构函数的执行顺序为: 先调用派生类的析构函数, 后调用基类的析构函数。



从范例 11-6 和执行结果读者可以看出，派生类和基类的构造函数的执行顺序与其析构函数的执行顺序正好相反。

11.4 多重继承

以上介绍的继承都是针对派生类只有一个基类的情况，这种情况称为单一继承。而当一个派生类具有多个基类时，称这种派生为多重继承。

11.4.1 二义性问题

在多重继承中，需要解决的主要问题是标识不唯一，即二义性问题。例如，当在派生类继承的这多个基类中有同名成员时，派生类中就会出现来自不同基类的同名成员，就出现了标识不唯一（二义性）的情况，这在程序中是不允许的。

【范例 11-7】多重继承中的二义性问题。该范例包含了多重继承中的二义性问题，其代码如代码清单 11-7 所示。

代码清单 11-7

```

1  class base1                                //定义基类 base1
2  {
3  public:                                    //定义公有成员
4      int x;
5      int a();                               //声明公有成员函数
6      int b();
7      int b(int);
8      int c();
9  };
10 class base2                                //定义基类 base2
11 {
12     int x;                                  //定义私有成员
13     int a();
14 public:                                    //定义公有成员
15     float b();
16     int c();
17 };
18 class derived:base1,base2                  //类 derived 私有继承于类 base1 和 base2
19 {                                           //无成员
20     void d(derived &e)                     //定义函数调用成员
21     {
22         e.x=10;                            //错误，不知道 x 是从哪个基类继承来的，有二义性
23         e.a();                              //错误，有二义性
24         e.b();                              //错误，有二义性
25         e.c();                              //错误，有二义性
26     }
27     void main()
28     {
29         derived ob;                         //创建对象
30     }

```

【运行结果】在 Visual C++ 中输入上述代码经编译后，系统运行结果如图 11-8 所示。

【范例解析】上述代码中的类 derived 继承于类 base1 和类 base2，其中类 base1 和类 base2 都含有成员 x 和函数 a()、b() 和 c()。在主函数 main() 中创建 derived 的对象 ob 时，编译系统无法识别上述语句第 22~25 行代码中的变量和函数分别继承于哪个类，因此导致了二义性问题。

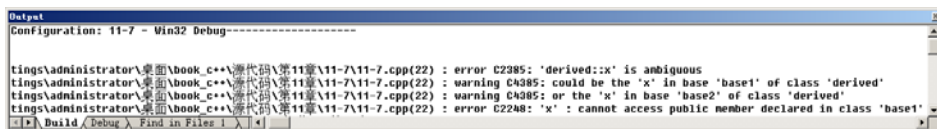


图 11-8 二义性问题

注意 多重继承中，派生类由多个基类派生时，基类之间以逗号“,”隔开，且其每个基类前都必须指明继承方式，否则默认为私有继承，如上述代码的第 18 行。

解决这个问题的办法有三种：

- 使用作用域运算符“::”。
- 使用同名覆盖的原则。
- 使用虚函数。

关于虚函数的有关问题将在后面讨论，这里介绍前两种方法。

1. 使用作用域运算符

如果派生类的基类之间没有继承关系，同时又没有共同的基类，则在引用同名成员时，可在成员名前加上类名和作用域运算符“::”来区别来自不同基类的成员。例如，将范例 11-7 中的函数 `d(derived &e)` 改写如下。

```
void d(derived &e)
{
    e.base1::x=10;
    e.base2::a();
    e.base2::b();
    e.base1::c();
}
```

2. 使用同名覆盖的原则

在派生类中重新定义与基类中同名的成员（如果是成员函数，则参数表也要相同，参数不同的情况为重载）以隐蔽掉基类的同名成员，在引用这些同名的成员时，就是使用派生类中的函数，这样就不会出现二义性的问题了。

【范例 11-8】使用作用域运算符和同名覆盖解决二义性问题。该范例定义的派生类和基类中都包含同一个函数，请读者观察其是如何调用的，代码如代码清单 11-8 所示。

代码清单 11-8

```
1  #include<iostream.h>
2  class base                                //定义基类 base
3  {
4  public:                                    //定义基类公有成员
5      int x;
6      void show()                          //定义基类公有成员函数
7      {
8          cout<<"This is base, x= "<<x<<endl;
9      };
10 };
11 class derived:public base                 //类 derived 公有继承于类 base
12 {
13 public:
14     int x;                                //同名数据成员
15     void show()                          //同名成员函数
16     {
```



```

17         cout<<"This is derived, x= "<<x<<endl;
18     }
19 };
20 void main()
21 {
22     derived ob;
23     ob.x=5;                //用同名覆盖原则引用派生类数据成员
24     ob.show();             //用同名覆盖原则引用派生类成员函数
25     ob.base::x=12;         //用作用域运算符访问基类成员
26     ob.base::show();       //用作用域运算符访问基类成员
27 }

```

【运行结果】在 Visual C++中执行上述代码，其运行结果如图 11-9 所示。

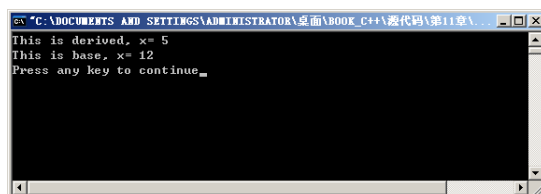


图 11-9 解决二义性问题

【范例解析】上述代码中，基类 base 和派生类 derived 都含有成员变量 x 和成员函数 show()，在主函数 main()中由派生类 derived 创建对象 ob 后，通过该对象用同名覆盖原则可引用派生类的成员，而通过作用域运算符可访问基类成员。



警告 这里的派生类与基类的继承方式必须为 public（公有继承），即第 11 行代码 class derived:public base 中的 public 是不能省略的，否则将不能通过编译。

11.4.2 声明多重继承

前面介绍了，多重继承就是一个派生类具有多个基类，这种情况在现实生活中也是常见的。比如，海洋中的鲸鱼既有哺乳动物的特征，也有鱼的特征，如图 11-10 所示。

C++中，多重继承声明的一般形式为：

```

class <派生类名>:<派生方式 1><基类名 1>,...,<派生方式 n><基类名 n>
{
    派生类成员声明;
}

```

其中，冒号后面的部分称为基类表，它们之间用逗号分开。派生方式规定了派生类以何种方式继承基类成员，仍为 private、protected 和 public 三类。在具体的程序中使用多重继承时应注意如下两个事项：

- 多重继承中，各种派生方式对于基类成员在派生类中的访问权限与单继承的规则相同。
- 在使用多继承时，对基类成员的访问应无二义性，如果有，则应使用 11.3 节介绍的方法予以解决。

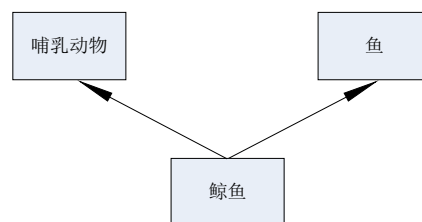


图 11-10 多重继承

【范例 11-9】多重继承的声明和使用。该范例定义了两个基类，并定义的派生类继承于这两个基类，读者应注意观察其成员的属性，代码如代码清单 11-9 所示。

代码清单 11-9

```

1  #include <iostream.h>
2  class base1                                //定义基类 base1
3  {
4  public:                                    //定义基类成员
5      int x;
6      void print()                          //定义基类公有成员函数
7      {
8          cout<<"class base1: x= "<<x<<endl;
9      }
10 };
11 class base2                                //定义基类 base2
12 {
13 public:                                    //定义基类成员
14     int x;
15     void print()
16     {
17         cout<<"class base2: x= "<<x<<endl;
18     }
19 };
20 class derived:public base1,public base2    //多重公有继承于 base1 和 base2
21 {
22 public:                                    //定义派生类成员
23     int x;
24     void print()                          //定义派生类成员函数
25     {
26         cout<<"class derived: x= "<<x<<endl;
27     }
28 };
29 void main()
30 {
31     derived ob;                            //创建对象
32     ob.x=10;                              //调用派生类成员
33     ob.print();
34     ob.base1::x=20;                       //调用基类 base1 成员
35     ob.base1::print();
36     ob.base2::x=30;                       //调用基类 base2 成员
37     ob.base2::print();
38 }

```

【运行结果】在 Visual C++中执行上述代码，其运行结果如图 11-11 所示。

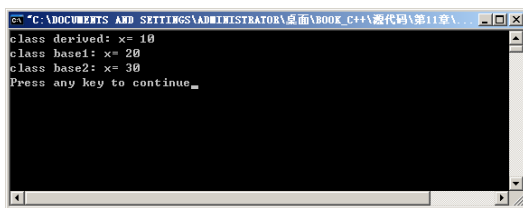


图 11-11 多重继承的声明

【范例解析】上述代码中，类 `derived` 公有继承于类 `base1` 和类 `base2`，其是一个多重继承。在主函数 `main()` 中创建一个派生类的对象后，通过该对象访问派生类中的成员与普通访问相同，而访问基类中的成员，如出现二义性则需使用作用域运算符来访问。

提示 使用作用域运算符来指明对象调用的某个成员属于哪个基类是很直观的方法，因此其在多重继承中使用非常广泛。



11.4.3 多重继承的构造函数和析构函数

与普通继承类似的，多重继承时，也涉及基类成员、对象成员和派生类成员的初始化问题，因此，必要时也要定义构造函数和析构函数。一般来说，C++中声明多继承构造函数的一般形式为：

```
<派生类名>::<派生类名>(参数总表)::基类名1(参数表1),...,基类名n(参数表n),对象成员名1(对象成员参数表1),...,对象成员名m(对象成员参数表m)
{
    //派生类新增成员的初始化语句
}
```

其中，需要注意如下几个事项：

- 派生类的构造函数名与派生类名相同。
- 参数总表列出初始化基类成员数据、新增对象成员数据和派生类新增成员数据所需要的参数。
- 冒号后列出需要使用参数进行初始化的所有基类的名字和所有对象成员的名字及各自的参数表，它们之间用逗号分开。对于使用默认构造函数的基类或对象成员，可以不给出类名或对象名及参数表。
- 多继承析构函数的声明方法与单继承的相同。
- 多重继承的构造函数和析构函数具有与单继承构造函数和析构函数相同的性质和特性。



注意 多重继承构造函数和析构函数的执行顺序与单继承的相同，但应强调的是，基类之间的执行顺序是严格按照声明时从左到右的顺序来执行的，与它们在定义派生类构造函数中的次序无关。

【范例 11-10】多重继承的构造函数和析构函数。该范例定义了多重继承的派生类的构造函数和析构函数，代码如代码清单 11-10 所示。

代码清单 11-10

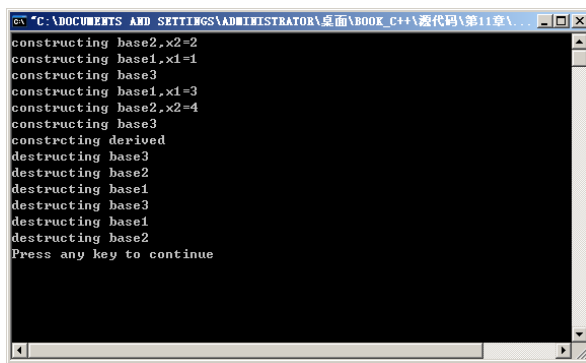
```
1  #include <iostream.h>
2  class base1                                //定义基类 base1
3  {
4      int x1;
5  public:
6      base1(int y1)                          //定义 base1 的构造函数
7      {
8          x1=y1;
9          cout<<"constructing base1,x1="<<x1<<endl;
10     }
11     ~base1()                               //定义 base1 的析构函数
12     {
13         cout<<"destructing base1"<<endl;
14     }
15 };
16 class base2                                //定义基类 base2
17 {
18     int x2;
19 public:
20     base2(int y2)                          //定义 base2 的构造函数
21     {
22         x2=y2;
23         cout<<"constructing base2,x2="<<x2<<endl;
24     }
25     ~base2()
```

```

26     {
27         cout<<"destructing base2"<<endl;
28     }
29 };
30 class base3                                //定义基类 base3
31 {
32     int x3;
33 public:
34     base3()                                //定义 base3 的构造函数
35     {
36         cout<<"constructing base3"<<endl;
37     }
38     ~base3()                               //定义 base3 的析构函数
39     {
40         cout<<"destructing base3"<<endl;
41     }
42 };
43 class derived:public base2,public base1,public base3
                                         //定义派生类
44 {
45 private:
46     base1 ob1;                             //创建基类对象
47     base2 ob2;
48     base3 ob3;
49 public:
50     derived(int x,int y,int z,int v):base1(x),base2(y),ob1(z),ob2(v)
                                         //派生类的构造函数
51     {
52         cout<<"constrcting derived"<<endl;
53     }
54 };
55 void main()
56 {
57     derived ob(1,2,3,4);                  //创建派生类对象
58 }

```

【运行结果】在 Visual C++ 中执行上述代码，其运行结果如图 11-12 所示。



```

C:\DOCUMENTS AND SETTINGS\ADMINISTRATOR\桌面\BOOK_C++\源代码\第11章\...
constructing base2,x2=2
constructing base1,x1=1
constructing base3
constructing base1,x1=3
constructing base2,x2=4
constructing base3
constructing derived
destructing base3
destructing base2
destructing base1
destructing base3
destructing base1
destructing base2
Press any key to continue

```

图 11-12 多重继承的构造函数和析构函数

【范例解析】上述代码中，类 `derived` 公有继承于类 `base1`、类 `base2` 和类 `base3`，派生类的构造函数为输出“constrcting derived”的信息。在主函数 `main()` 中创建派生类的对象 `ob` 时，其先调用 `base2` 的构造函数，在调用 `base1` 的构造函数，最后调用 `base3` 的构造函数，这是因为在声明派生类的第 43 行代码时，其声明顺序为 `base2`、`base1`、`base3`。在释放对象调用析构函数时，其顺序正好与构造函数的顺序相反。



创建派生类的对象时，首先根据派生类定义的继承于基类的顺序执行基类的构造函数，再执行派生类的构造函数。

11.5 虚基类

在介绍多重继承的概念时已经知道，多重继承中，要引用派生类的成员时，先是在派生类自身的作用域内寻找，如果找不到，再到基类中寻找。这时，如果这些基类又有一个共同的基类，派生类访问这个公共的成员时，就有可能由于同名成员的问题而发生二义性，此时就需要使用到本节要介绍的虚基类。

11.5.1 虚基类的引入

引入虚基类的主要原因是为了解决基类中由于同名成员的问题而产生的二义性问题。例如，下面范例就产生二义性。

【范例 11-11】虚基类的引入。该范例在实现多重继承的时候出现了二义性问题，请读者观察其是如何出现的，其代码如代码清单 11-11 所示。

代码清单 11-11

```

1  #include<iostream.h>
2  class base                                //定义基类
3  {
4  protected:
5      int x;
6  public:
7      base()                                //定义基类构造函数
8      {
9          x=1;
10     };
11 };
12 class base1:public base                    //定义 base 的派生类 base1
13 {
14 public:
15     base1()                                //定义派生类 base1 的构造函数
16     {
17         cout<<"constructing base1,x="<<x<<endl;
18     }
19 };
20 class base2:public base                    //定义 base 的派生类 base2
21 {
22 public:
23     base2()                                //定义派生类 base2 的构造函数
24     {
25         cout<<"constructing base2,x="<<x<<endl;
26     }
27 };
28 class derived:public base1,public base2    //定义多重继承于 base1 和 base2 的派生类
29 {
30 public:
31     derived()                                //定义派生类 derived 的构造函数
32     {
33         cout<<"constructing derived x="<<x<<endl;
34     }
35 };
36 void main()

```

```

37 {
38     derived obj;           //创建类 derived 的对象 obj
39 }

```

【运行结果】在 Visual C++ 中输入上述代码经编译后，系统会返回如图 11-13 所示的错误提示信息。



图 11-13 二义性的产生

【范例解析】从上述编译结果读者可以看出，这是一个有问题的程序。因为表面看起来类 base1 和类 base2 是从同一个基类 base 派生来的，但其对应的却是基类 base 的两个不同的拷贝。因此，当派生类 derived 要访问变量 x 时不知从哪条路径去寻找，从而引发了二义性问题。

注意 一般来说，在多重继承中，当派生类有多条路径去访问基类，即非虚基类时，编译系统会给出错误信息。

在多重继承中，非虚基类的类层次图如图 11-14 所示。虚基类就是为了解决这个问题而引入的，其具体的做法是将公共基类声明为虚基类，这样这个公共基类就只有一个拷贝而不会出现二义性了。虚基类的类层次图如图 11-15 所示。

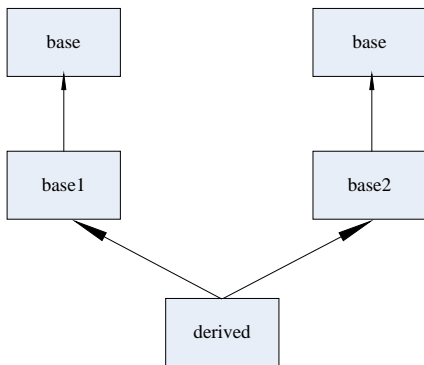


图 11-14 非虚基类的类层次图

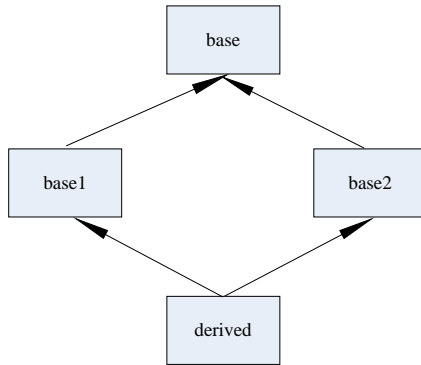


图 11-15 虚基类的类层次图

11.5.2 定义虚基类

虚基类的声明是在派生类的声明过程中进行的，其声明的一般形式为：

```
class<派生类名>:virtual<派生方式><基类名>
```

虚基类关键字的作用范围和派生方式与一般派生类的声明一样，只对紧跟其后的基类起作用。声明了虚基类以后，虚基类的成员在进一步派生过程中和派生类一起维护同一个内存拷贝。

【范例 11-12】虚基类的定义。该范例将上述代码清单 11-11 使用虚基类改写如下，其使用虚基类解决二义性问题，其实现代码如代码清单 11-12 所示。



代码清单 11-12

```

1  #include<iostream.h>
2  class base                                //定义基类
3  {
4  protected:                                //定义基类保护成员
5      int x;
6  public:
7      base()                                //定义基类构造函数
8      {
9          x=1;
10     }
11 };
12 class base1:virtual public base            //定义 base 为虚基类
13 {
14 public:
15     base1()                                //定义派生类 base1 的构造函数
16     {
17         cout<<"constructing base1,x="<<x<<endl;
18     }
19 };
20 class base2:virtual public base            //定义 base 为虚基类
21 {
22 public:
23     base2()                                //定义派生类 base2 的构造函数
24     {
25         cout<<"constructing base2,x="<<x<<endl;
26     }
27 };
28 class derived:public base1,public base2    //定义多重继承于 base1 和 base2 的派生类
29 {
30 public:
31     derived()                                //定义派生类 derived 的构造函数
32     {
33         cout<<"constructing derived x="<<x<<endl;
34     }
35 };
36 void main()
37 {
38     derived obj;                            //创建类 derived 的对象 obj
39 }

```

【运行结果】在 Visual C++中执行上述代码，其运行结果如图 11-16 所示。

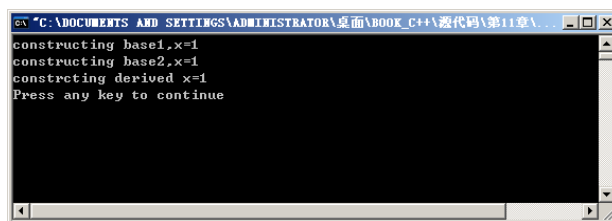


图 11-16 虚基类的定义和使用

【范例解析】上述范例中，由于把公共基类 base 声明为类 base1 和类 base2 的虚基类，所以由类 base1 和类 base2 派生的类 derived 只有一个基类 base，从而消除了二义性。



注意 虚基类的定义只需要在基类的声明中，在继承方式前添加 virtual 关键字即可，编译系统将自动消除二义性。

11.5.3 虚基类的构造函数和初始化

虚基类的初始化与一般的多继承的初始化在语法上是一样的，但构造函数的执行顺序不同：

- 虚基类的构造函数的执行在非虚基类的构造函数之前。
- 若同一层次中包含多个虚基类，这些虚基类的构造函数按对它们说明的先后次序执行。
- 若虚基类由非虚基类派生而来，则仍然先执行基类的构造函数，再执行派生类的构造函数。

【范例 11-13】虚基类的构造函数。该范例定义了 4 个类，其中类 base1 和类 base2 有共同的基类 base，将该基类声明为虚基类，读者应注意查看其构造函数执行的顺序，其代码如下清单 11-13 所示。

代码清单 11-13

```

1  #include <iostream.h>
2  class base                                //定义基类 base
3  {
4  protected:                                //定义基类保护成员
5      int x;
6  public:
7      base(int x1)                            //定义基类构造函数
8      {
9          x=x1;
10         cout<<"constructing base,x="<<x<<endl;
11     }
12 };
13 class base1:virtual public base            //定义基类 base1 为虚基类
14 {
15     int y;                                    //定义虚基类私有成员
16 public:
17     base1(int x1,int y1):base(x1)           //定义虚基类 base1 的构造函数
18     {
19         y=y1;
20         cout<<"constructing base1,y="<<y<<endl;
21     }
22 };
23 class base2:virtual public base            //定义基类 base2 为虚基类
24 {
25     int z;                                    //定义虚基类私有成员
26 public:
27     base2(int x1,int z1):base(x1)           //定义虚基类 base2 的构造函数
28     {
29         z=z1;
30         cout<<"constructing base2,z="<<z<<endl;
31     }
32 };
33 class derived:public base1,public base2     //定义多重继承的派生类
34 {
35     int xyz;                                    //定义派生类私有成员
36 public:
37     derived(int x1,int y1,int z1,int xyz1):base(x1),base1(x1,y1),base2(x1,z1)
                                                //派生类构造函数
38     {
39         xyz=xyz1;
40         cout<<"constructing derived xyz="<<xyz<<endl;
41     }
42 };
43 void main()
44 {

```



```
45         derived obj(1,2,3,4);           //创建派生类对象
46     }
```

【运行结果】在 Visual C++ 中执行上述代码，其运行结果如图 11-17 所示。

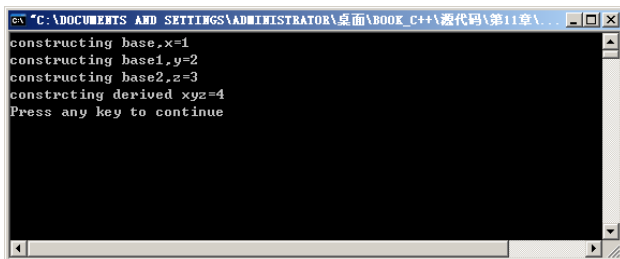


图 11-17 虚基类的构造函数

【范例解析】读者可以看出，范例 11-13 中虚基类 base 的构造函数只执行了一次。这是因为当派生类 derived 调用了虚基类 base 的构造函数之后，类 base1 和类 base2 对虚基类 base 构造函数调用被忽略了。这是初始化虚基类和初始化非虚基类的不同。



注意 在实际使用虚基类时应注意如下几个问题。虚基类的关键字 virtual 与派生方式的关键字 private、protected 和 public 的书写位置无关紧要，可以先写虚基类的关键字，也可以先写派生方式的关键字；一个基类在作为某些派生类的虚基类的同时也可作为另一些派生类的非虚基类；虚基类构造函数的参数必须由最新派生出来的类负责初始化，即使不是直接继承也应如此。

11.6 小结

本章主要介绍了 C++ 面向对象程序设计的一个重要特征——继承的概念和使用。本章一开始就开门见山地介绍了继承和派生的概念，接着详细讲解了 C++ 中的三种继承：公有继承、私有继承和保护继承。这三种不同的继承方式，其派生出来的派生类的成员属性也是不同的，因此，对其不同成员的访问控制机制本章也做了详细的讲解。类的一个重要特征是构造函数和析构函数，针对派生类也是如此，本章通过具体的范例讲解了派生类的构造函数、析构函数及其执行顺序。此外，本章对多重继承、虚基类都做了较为详细的讲解。

11.7 习题

1. 编写程序设计一个汽车类 vehicle，包含的数据成员有：车轮个数 wheels 和车重 weight。小车类 car 是它的私有派生类，其中包含载人数 passenger_load。卡车类 truck 是 vehicle 的私有派生类，其中包含载人数 passenger_load 和载重量 payload，每个类都有相关数据的输出方法。

【解答】该习题主要考查单继承的实现。在程序实现时，vehicle 类是基类，由它派生出 car 类和 truck 类。由于 car 和 truck 类都有车轮个数 wheels 和车重 weight 这两个公共属性，因此将公共属性和方法放在 vehicle 类中，其他具体属性放在各自派生类中。其简要的实现代码如下所示。

```
class vehicle                               // 定义汽车类
{
protected:
    int wheels;                             // 车轮数
    float weight;                           // 重量
```

```

public:
vehicle(int wheels,float weight);
int get_wheels();
float get_weight();
float wheel_load();
void show();
};
class car:public vehicle           // 定义小车类
{
int passenger_load;               // 载人数
public:
car(int wheels,float weight,int passengers=4);
int get_passengers();
void show();
};
class truck:public vehicle        // 定义卡车类
{
int passenger_load;              // 载人数
float payload;                   // 载重量
public:
truck(int wheels,float weight,int passengers=2,float max_load=24000.00);
int get_passengers();
float efficiency();
void show();
};

```

2. 分析以下程序的执行结果。

```

#include<iostream.h>
class base
{
public:
    base(){cout<<"constructing base class"<<endl;}
    ~base(){cout<<"destructing base class"<<endl; }
};
class subs:public base
{
public:
    subs(){cout<<"constructing sub class"<<endl;}
    ~subs(){cout<<"destructing sub class"<<endl;}
};
int main()
{
    subs s;
}

```

【解答】该习题主要考查单继承情况下构造函数和析构函数的调用顺序。此处 base 为基类，subs 为派生类。读者应知道，在单继承情况下，首先调用基类的构造函数，随后调用派生类的构造函数，析构函数的调用顺序则正好相反。因此，该习题的输出应为：

```

constructing base class
constructing sub class
destructing sub class
destrcuting base class

```

3. 编写一个 C++ 程序，假设图书馆的图书包含书名、编号和作者等 3 个属性，读者包含姓名和借书证属性，每位读者最多可借 5 本书，编写程序列出某读者的借书情况。

【解答】该习题主要考查类的继承设计。此处可以设计一个类，从它派生出书类 book 和读者类 reader，在 reader 类中有一个 rentbook() 成员函数用于借阅图书。其简要的实现代码如下所示。

```

class object
{

```




```

        char name[20];
        int no;
public:
    object(){}
    object(char na[],int n) ;
    void show() ;
};
class book:public object
{
    char author[10];
public:
    book(){}
    book(char na[],int n,char auth[]):object(na,n) ;
    void showbook();
};
class reader:public object
{
    book rent[5];
    int top;
public:
    reader(char na[],int n):object(na,n){top=0;}
    void rentbook(book &b) ;
    void showreader() ;
};

```

4. 分析以下程序的执行结果。

```

#include<iostream.h>
class A
{
public:
    int n;
};
class B:virtual public A{};
class C:virtual public A{};
class D:public B,public C
{
    int getn(){return B::n;}
};
void main()
{
    D d;
    d.B::n=10;
    d.C::n=20;
    cout<<d.B::n<<" "<<d.C::n<<endl;
}

```

【解答】该习题主要考查虚基类的实现。上述程序中，D 类是从类 B 和类 C 派生的，而类 B 和类 C 又都是从类 A 派生的，但这是虚继承关系，即是虚基类。因此，类 B 和类 C 共用一个副本，所以对于类 D 的对象 d，d.B::n 与 d.C::n 是一个成员。因此，上述程序段输出为： 20，20。

5. 设计一个圆类 circle 和一个桌子类 table，另设计一个圆桌类 roundtable，它是从前两个类派生的，要求输出一个圆桌的高度、面积和颜色等数据。

【解答】该习题主要考查多重继承的类的设计。该习题中，circle 类包含私有数据成员 radius 和求圆面积的成员函数 getarea(); table 类包含私有数据成员 height 和返回高度的成员函数 getheight()。roundtable 类继承所有上述类的数据成员和成员函数，添加了私有数据成员 color 和相应的成员函数。其简要的实现代码如下所示。

```

class circle
{
    double radius;
public:

```

```

        circle(double r) { radius=r; }
        double getarea() { return radius*radius*3.14; }
};
class table
{
    double height;
public:
    table(double h) { height=h; }
    double getheight() { return height; }
};
class roundtable : public table,public circle
{
    char *color;
public:
    roundtable(double h, double r, char c[]) : circle (r) , table (h)
    {
        color=new char[strlen(c)+1];
        strcpy (color, c);
    }
    char *getcolor() { return color; }
};

```

6. 设计一个日期类 `Date` 和一个时间类 `Time`，由这两个类派生出日期时间类 `TimeDate`。在创建对象时将指定的日期时间调用不同类的成员函数进行输出，结果如图 11-18 所示。

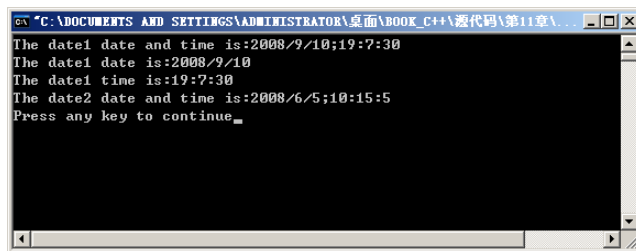


图 11-18 综合练习

【解答】该程序的日期时间类 `TimeDate` 公有继承于类 `Date` 和类 `Time`，需先定义基类 `Date` 和 `Time`。在主函数 `main()` 中创建两个对象 `date1` 和 `date2`，其中对象 `date2` 是在创建时即进行了初始化，`date1` 则是通过调用其基类的成员函数进行赋值，将这两个对象的日期时间字符串、日期和时间字符串输出，其分别调用了不同类的不同成员函数来实现。其简要的实现代码如下所示。

```

typedef char string80[80]; //重定义字符串数据类型
class Date //定义类
{
public: //定义公有成员
    Date() {} //声明构造函数
    Date(int y, int m, int d) //重载构造函数
    {
        SetDate(y, m, d); //调用函数
    }
    void SetDate(int y, int m, int d) //定义公有成员函数
    {
        Year = y; //给成员变量赋初值
        Month = m;
        Day = d;
    }
    string80& GetStringDate(string80 &Date) //定义公有成员函数
    {
        sprintf(Date, "%d/%d/%d", Year, Month, Day); //输出字符串
    }
};

```



```

        return Date;                                //返回结果
    }
    protected:                                     //定义保护成员
        int Year, Month, Day;                       //定义整型变量
};
class Time                                          //定义类 Time
{
public:                                             //定义公有成员
    Time() {}                                       //定义构造函数
    Time(int h, int m, int s)                     //重载构造函数
    {
        SetTime(h, m, s);                         //调用设置时间函数
    }
    void SetTime(int h, int m, int s)              //定义公有成员函数
    {
        Hours = h;                                //给成员变量赋初值
        Minutes = m;
        Seconds = s;
    }
    string80& GetStringTime(string80 &Time)        //定义公有成员函数
    {
        sprintf(Time, "%d:%d:%d", Hours, Minutes, Seconds);
                                                //输出字符串
        return Time;                               //返回结果
    }
protected:                                       //定义保护成员
    int Hours, Minutes, Seconds;                  //定义整型变量
};
class TimeDate:public Date, public Time           //定义派生类
{
public:                                           //定义公有成员
    TimeDate():Date() {}                         //声明派生类构造函数
    TimeDate(int y, int mo, int d, int h, int mi, int s):Date(y, mo, d), Time(h, mi, s) //重载构造函数
    {}
    string80& GetStringDT(string80 &DTstr)         //定义派生类成员函数
    {
        sprintf(DTstr, "%d/%d/%d;%d:%d:%d", Year, Month, Day, Hours, Minutes,
Seconds)
        return DTstr;                             //返回结果
    }
};
void main()
{
    TimeDate date1, date2(2008, 6, 5, 10, 15, 05); //创建派生类对象
    string80 DemoStr;                               //定义字符串数据类型
    date1.SetDate(2008, 9, 10);                     //调用基类成员函数
    date1.SetTime(19, 07, 30);                       //调用设置时间函数
    cout<<"The date1 date and time is:"<<date1.GetStringDT(DemoStr)<<endl; //调用派生类成员函数
    cout<<"The date1 date is:"<<date1.GetStringDate(DemoStr)<<endl;         //调用成员函数输出结果
    cout<<"The date1 time is:"<<date1.GetStringTime(DemoStr)<<endl;
    cout<<"The date2 date and time is:"<<date2.GetStringDT(DemoStr)<<endl;
}

```

第 12 章 多态

第 11 章介绍的继承是 C++ 面向对象程序设计的一个重要特征，而本章要讲解的多态也是面向对象的一个重要特征。类的多态特性是支持面向对象语言最主要的特性，有过非面向对象语言开发经历的人，通常对这一章节的内容会觉得不习惯，因为很多人会错误地认为，支持类封装的语言就是支持面向对象的，其实不然，比如 Visual Basic 是典型的非面向对象的开发语言，但是它的确支持类，因此支持类并不能说明就是支持面向对象。只有能够解决多态问题的语言，才是真正支持面向对象的开发的语言。因此，多态性（polymorphism）是面向对象编程的基本特征之一。

以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为一个学习的参照。

- 理解多态的概念。
- 熟练掌握 C++ 中多态的实现方法。
- 熟练掌握虚函数的定义及其使用。
- 掌握纯虚函数和抽象类。

12.1 多态

在面向对象的概念中，多态性是指不同对象接收到相同消息时，根据对象类的不同而产生不同的动作。多态性提供了同一个接口可以用多种方法进行调用的机制，从而可以通过相同的接口访问不同的函数。具体地说，多态性就是同一个函数名称，作用在不同的对象上将产生不同的操作。

12.1.1 什么是多态

简单来说，多态就是“一个接口，多种实现”，就是同一种事物表现出的多种形态。例如，一个人跟随旅游团去北京，其只要跟着旅游团去即可，至于如何去，是坐火车、坐飞机还是其他方式，这由旅游团来实现，如图 12-1 所示。

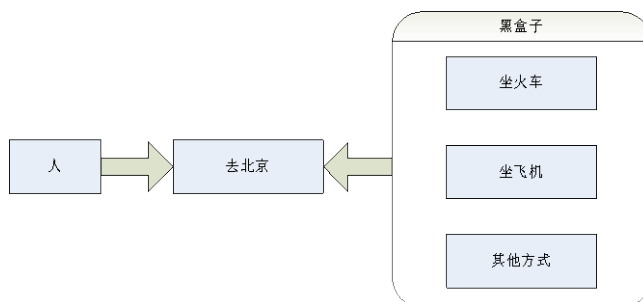


图 12-1 多态的含义

图 12-1 中的黑盒子即为“去北京”这个接口所表现出来的多种形态，程序员只需调用该接口，至于其如何实现，则不用管它。面向对象的多态性可以严格的分为四类：重载多态、强制多态、包含多态和参数多态。前面两种统称为专用多态，后面两种统称为通用多态。

包含多态是指定义于不同类中的同名成员函数的多态行为，主要通过虚函数来实现。参数



多态与类模板相关联，其是一个可以参数化的模板，其中包含的操作所涉及的类型必须用类型参数进行实例化。这样，由类模板实例化的各类都具有相同的操作，而操作对象的类型却各不相同。

多态从实现的角度来讲可以划分为两类，编译时的多态和运行时的多态。前者是在编译的过程中确定了同名操作的具体操作对象，而后者则是在程序运行过程中才动态地确定操作所针对的具体对象。这种确定操作的具体对象的过程就是联编。

联编是指计算机程序自身彼此关联的过程，即把一个标识符名和一个存储地址联系在一起的过程。用面向对象的术语讲，就是把一条消息和一个对象的方法相结合的过程。按照联编进行阶段的不同，可以分为两种不同的联编方法：静态联编和动态联编，这两种联编过程分别对应着多态的两种实现方式。



提示 在本书中，主要从实现的角度来讲解多态，即分为编译时的多态和运行时的多态来介绍多态的实现。

12.1.2 多态的作用

事实上，多态也是人类思维方式的一种直接模拟。比如，一个对象中有很多求两个数最大值的行，虽然可以针对不同的数据类型，写很多不同名称的函数来实现，但事实上，它们的功能几乎完全相同。这时，就可以利用多态的特征，用统一的标识来完成这些功能。这样，就可以达到类的行为的再抽象，进而统一标识，减少程序中标识符的个数。

例如，图 12-2 表示的是求两个数最大值的多种表示，其实这些表示都可以通过一个统一的标识来表示。

图 12-2 中，在 C++ 中求两个数的最大值需要分多种情况：求两个整数的最大值、求两个浮点数的最大值和求两个字符的最大值。这些都需要不同名称的函数如 `maxi`、`maxf` 和 `maxc` 来实现，而多态就是将这些不同的函数名称统一用 `max` 来表示。

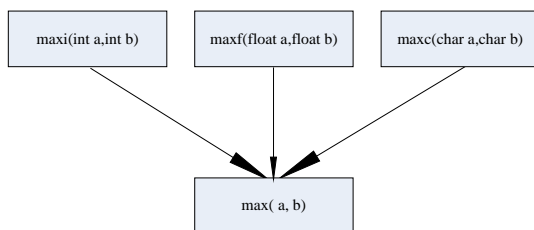


图 12-2 多态的含义

总的来说，多态性提供了把接口与实现分开的另一种方法，提高了代码的组织性和可读性，更重要的是提高了软件的可扩充性。



注意 多态其实是一种行为的封装，程序员只需知道自己所操纵的对象所能够做的事情（接口），那么就可以在需要的时候叫它去做，具体怎么做由它自己去决定，程序员不需要知道而且没有必要知道。

12.1.3 多态的引入

为了让读者更好地理解多态在实际程序中的应用，下面给出一个范例，引出要使用多态的原因。

【范例 12-1】多态的引入。范例是一个没有使用多态的例子，请读者观察其输出应该是什么，其代码如代码清单 12-1 所示。

代码清单 12-1

```
1  #include <iostream.h>
2  class animal                      //定义基类 animal
```

```

3  {
4  public:
5      void sleep()                //定义成员函数
6      {
7          cout<<"animal sleep"<<endl;
8      }
9      void breathe()              //定义成员函数
10     {
11         cout<<"animal breathe"<<endl;
12     }
13 };
14 class fish:public animal          //定义 animal 的公有派生类 fish
15 {
16 public:
17     void breathe()                //定义同名成员函数
18     {
19         cout<<"fish bubble"<<endl;
20     }
21 };
22 void main()
23 {
24     fish fh;                      //创建对象
25     animal *pAn=&fh;              //定义对象指针指向对象 fh
26     pAn->breathe();               //调用 breathe 函数
27 }

```

【运行结果】在 Visual C++ 中新建一个【C++ Source File】文件，将上述代码输入，编译无误后执行，其运行结果如图 12-3 所示。

【范例解析】上述代码中，在主函数 main() 中首先定义了一个 fish 类的对象 fh，接着定义了一个指向 animal 类的指针变量 pAn，将 fh 的地址赋给了指针变量 pAn，然后利用该变量调用 pAn->breathe()。许多读者往往将这种情况和 C++ 的多态性搞混淆，认为 fh 实际上是 fish 类的对象，应该是调用 fish 类的 breathe()，然后输出“fish bubble”，但结果却不是这样的。

这是因为 C++ 编译器在编译的时候，要确定每个对象调用的函数的地址，这称为早期绑定 (early binding)。当将 fish 类的对象 fh 的地址赋给 pAn 时，C++ 编译器进行了类型转换，此时 C++ 编译器认为变量 pAn 保存的就是 animal 对象的地址。当在 main() 函数中执行 pAn->breathe() 时，调用的当然就是 animal 对象的 breathe 函数。此时，类 fish 创建的对象 fh 在内存中的存储如图 12-4 所示。

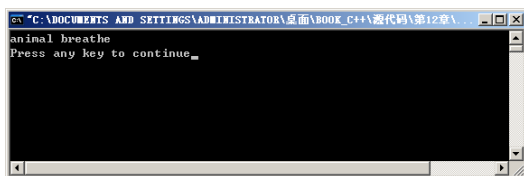


图 12-3 多态的引入



图 12-4 对象所占内存

在构造 fish 类的对象时，系统首先要调用 animal 类的构造函数去构造 animal 类的对象，然后才调用 fish 类的构造函数完成自身部分的构造，从而拼接出一个完整的 fish 对象。因此，当将 fish 类的对象转换为 animal 类型时，该对象就被认为是原对象整个内存模型的上半部分，也就是图 12-4 中的“animal 的对象所占内存”。那么当利用类型转换后的对象指针去调用它的方法时，当然也就是调用它所在的内存中的方法。因此，输出 animal breathe。



上述程序的运行结果并不是读者所希望看到的输出“fish bubble”信息，这个问题就可以通过本章介绍的多态来解决。

12.2 函数重载

前面提到了，由静态联编支持的多态性称为编译时的多态性或静态多态性，也就是说，确定同名操作的具体操作对象的过程是在编译过程中完成的。在 C++ 中，可以用函数重载和运算符重载来实现编译时的多态性。

由动态联编支持的多态性称为运行时的多态性或动态多态性，也就是说，确定同名操作的具体操作对象的过程是在运行过程中完成的。在 C++ 中，可以用继承和虚函数来实现运行时的多态性。本节将介绍通过函数重载来实现静态多态性，12.3 节将介绍通过虚函数来实现动态多态性，后续篇章将对运算符重载做详细讲解。

函数重载在前面的章节中也提到过，函数的重载也称多态函数，是实现编译时的多态性的形式之一。它使程序能用同一个名字来访问一组相关的函数，提高了程序的灵活性。函数重载时，函数名相同，但函数所带的参数个数或数据类型不同，编译系统会根据参数来决定调用哪个同名的函数。

面向对象程序设计中，函数的重载表现为两种情况：

- 第一种是参数个数或类型有所差别重载。
- 第二种是函数的参数完全相同但属于不同的类。

关于第一种情况的内容已在前面章节中作了详细介绍，此处主要介绍第二种情况的内容。当函数的参数完全相同但属于不同的类时，为了让编译程序能正确区分调用哪个类的同名函数，采用以下两种方法。

- 用对象名区别：在函数名前加上对象名来限制。例如，对象名.函数名。
- 用类名和作用域运算符::加以区别：在函数名前加“类名::”来限制。例如，类名::函数名。

【范例 12-2】用函数重载实现多态。该范例定义两个类，其中都包含同一个名称的函数，为了实现编译时的多态性，即让程序能通过同一名字来访问这个函数，代码如代码清单 12-2 所示。

代码清单 12-2

```
1  #include<iostream.h>
2  class point                                //定义基类 point
3  {
4      int x,y;
5  public:
6      point(int xx,int yy)                  //定义基类构造函数
7      {
8          x=xx;
9          y=yy;
10     }
11     double area()                          //定义基类的成员函数
12     {
13         return 0.0;
14     }
15 };
16 class circle:public point                  //派生类公有继承于基类
17 {
18     int r;
```

```

19 public:
20     circle(int xx,int yy,int rr):point(xx,yy) //定义派生类构造函数
21     {
22         r=rr;
23     }
24     double area() //同名函数
25     {
26         return 3.1416*r*r;
27     }
28 };
29 void main()
30 {
31     point pob(15,15); //创建基类对象
32     circle cob(20,20,10); //创建派生类对象
33     cout<<"pob.area()= "<<pob.area()<<endl; //调用基类成员函数
34     cout<<"cob.area()= "<<cob.area()<<endl; //调用派生类成员函数
35     cout<<"cob.point::area()= "<<cob.point::area()<<endl; //派生类对象调用基类成员函数
36 }

```

【运行结果】在 Visual C++ 中输入上述代码，运行结果如图 12-5 所示。

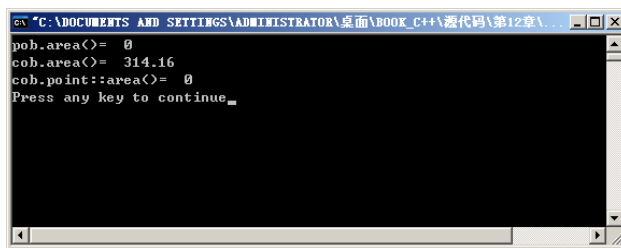


图 12-5 函数重载

【范例解析】上述代码中，类 circle 公有继承于类 point，其都定义了求面积的成员函数 area()。在主函数 main() 中创建了两个对象 pob 和 cob，各自调用其 area 函数，再使用 cob 对象调用类 point 的 area() 函数，其调用方法为：类名::函数名。



提示 通过上述函数重载方式，就实现了同一个函数名称实现不同的函数功能了，即实现了编译时的多态性，静态多态性。

12.3 虚函数

虚函数是重载的另一种形式，实现的是动态的重载，即函数调用与函数体之间的联系是在运行时才建立，也就是动态联编。前面的内容也提到了，虚函数是实现运行时的多态，即动态多态性的一个重要方式。

12.3.1 虚函数的引入

一般对象的指针之间没有联系，彼此独立，不能混用。但派生类是由基类派生出来的，它们之间有继承关系，因此，指向基类和派生类的指针之间也有一定的联系。如果使用不当，将会出现一些问题，比如下面的范例。

【范例 12-3】虚函数的引入。该范例没有使用虚函数，通过该范例读者可以看出程序中为什么要使用虚函数，代码如代码清单 12-3 所示。



代码清单 12-3

```

1  #include<iostream.h>
2  class base                                //定义基类 base
3  {
4  private:                                  //定义基类私有成员
5      int x,y;
6  public:
7      base(int xx=0,int yy=0)              //定义构造函数
8      {
9          x=xx;
10         y=yy;
11     }
12     void disp()                          //定义成员函数
13     {
14         cout<<"base: "<<x<<" " <<y<<endl;
15     }
16 };
17 class base1:public base                  //定义公有派生类 base1
18 {
19 private:                                  //定义派生类 base1 私有成员
20     int z;
21 public:
22     base1(int xx,int yy,int zz):base(xx,yy) //定义派生类的构造函数
23     {
24         z=zz;
25     }
26     void disp()                          //定义同名函数
27     {
28         cout<<"base1: "<<z<<endl;
29     }
30 };
31 void main()
32 {
33     base obj(3,4),*objp;                 //创建基类的对象和对象指针
34     base1 obj1(1,2,3);                   //创建派生类的对象
35     objp=&obj;                            //对象指针指向基类
36     objp->disp();                         //调用基类成员函数
37     objp=&obj1;                          //对象指针指向派生类
38     objp->disp();                         //调用派生类成员函数
39 }

```

【运行结果】在 Visual C++ 中输入上述代码，运行结果如图 12-6 所示。

【范例解析】读者可以看出，上述代码中虽然执行语句 `objp=&obj1` 之后，指针已经指向了对象 `obj1`，但其调用的函数仍然是基类对象的函数而不是派生类对象的函数，原本想通过使用对象指针来达到动态调用，即当指针指向不同的对象时执行不同操作的目的没有达到，出现这种现象的原因是调用的成员函数在编译时静态联编了。这种情况与范例 12-1 是类似的，其并没有输出预期的结果。

为解决这个问题，就要引入虚函数的概念。在介绍虚函数之前，先简单说明派生类对象指针使用时应注意的问题：

- 声明为指向基类对象的指针可以指向它的公有派生类的对象，但不允许指向它的私有派生类的对象。
- 允许声明为指向基类对象的指针指向它的公有派生类的对象，但不允许将一个声明为

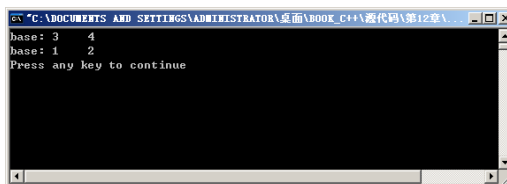


图 12-6 虚函数的引入

指向派生类对象的指针指向基类的对象。

- 声明为指向基类对象的指针，当其指向它的公有派生类的对象时，只能直接访问派生类中从基类继承下来的成员，不能直接访问公有派生类中定义的成员。要想访问其公有派生类中的成员，可将基类指针用显式类型转换方式转换为派生类指针。



注意 在 C++ 中引入虚函数的概念，主要是为了解决在调用既有继承关系的类的成员函数时能够正确地被执行。

12.3.2 定义虚函数

虚函数的定义是在基类中进行的，即把基类中需要定义为虚函数的成员函数声明为 `virtual`。当基类中的某个成员函数被声明为虚函数后，其就可以在派生类中被重新定义。在派生类中重新定义时，其函数原型，包括返回类型、函数名、参数个数和类型、参数的顺序都必须与基类中的原型完全一致。一般来说，虚函数定义的形式为：

```
virtual <函数类型><函数名>(参数表)
{
    函数体
}
```

【范例 12-4】虚函数的定义。该范例将代码清单 12-1 进行改写，使其得到预期的输出，只需将其中的一个函数定义为虚函数即可，其实现代码如代码清单 12-4 所示。

代码清单 12-4

```
1  #include <iostream.h>
2  class animal                                //定义基类 animal
3  {
4  public:
5      void sleep()                            //定义成员函数
6      {
7          cout<<"animal sleep"<<endl;
8      }
9      virtual void breathe()                 //定义虚函数
10     {
11         cout<<"animal breathe"<<endl;
12     }
13 };
14 class fish:public animal                    //定义 animal 的公有派生类 fish
15 {
16 public:
17     void breathe()                          //定义同名成员函数
18     {
19         cout<<"fish bubble"<<endl;
20     }
21 };
22 void main()
23 {
24     fish fh;                                //创建对象
25     animal *pAn=&fh;                        //定义对象指针指向对象 fh
26     pAn->breathe();                          //调用 breathe 函数
27 }
```

【运行结果】在 Visual C++ 中输入上述代码，运行结果如图 12-7 所示。

【范例解析】读者可以看到，将基类中的函数 `breathe()` 定义为虚函数后，即将上述代码的第 9 行加上了 `virtual` 关键字，再在主函数 `main()` 中定义对象指针指向类 `fish` 的对象 `fh`，调用



breathe()函数后，得到的结果就是预期的结果。



警告 虚函数的定义必须在基类中进行，即只有基类中的函数才能被定义为虚函数，在派生类中不能定义某个函数为虚函数。

事实上，当将 breathe()声明为 virtual 即虚函数时，编译器在编译的时候，发现 animal 类中有虚函数，此时编译器会为每个包含虚函数的类创建一个虚表，该表是一个一维数组，在这个数组中存放每个虚函数的地址。对于范例 12-4 的程序，animal 和 fish 类都包含了一个虚函数 breathe()，因此编译器会为这两个类都建立一个虚表，如图 12-8 所示。

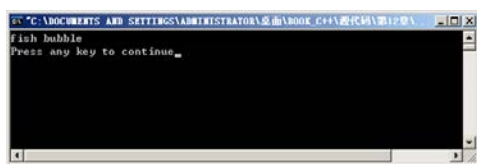


图 12-7 虚函数的定义

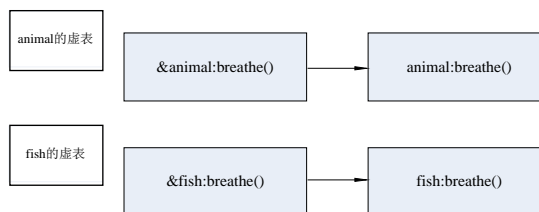


图 12-8 类 animal 和类 fish 的虚表

对于上述程序来说，当 fish 类的 fh 对象构造完毕后，其内部的虚表指针也就被初始化为指向 fish 类的虚表。在类型转换后，调用 pAn->breathe()，由于 pAn 实际指向的是 fish 类的对象，该对象内部的虚表指针指向的是 fish 类的虚表，因此最终调用的是 fish 类的 breathe()函数。

12.3.3 使用虚函数

事实上，读者可以看出，在派生类中被重新定义的基类中的虚函数，是函数重载的另一种形式，但它与函数重载又有如下的区别：一般的函数重载，要求其函数的参数或参数类型必须有所不同，函数的返回类型也可以不同，但重载一个虚函数时，要求函数名、返回类型、参数个数、参数的类型和参数的顺序必须与基类中的虚函数的原型完全相同。

- 如果仅返回类型不同，其余相同，则系统会给出错误信息。
- 如果函数名相同，而参数个数、参数的类型或参数的顺序不同，系统认为是普通的函数重载，虚函数的特性将丢失。

【范例 12-5】使用虚函数。该范例使用了虚函数，得到了用户预期希望得到的结果，其实现代码如代码清单 12-5 所示。

代码清单 12-5

```

1  #include<iostream.h>
2  class base                                //定义基类 base
3  {
4  private:                                  //定义基类私有成员
5      int x,y;
6  public:
7      base(int xx=0,int yy=0)                //定义构造函数
8      {
9          x=xx;
10         y=yy;
11     }
12     virtual void disp()                    //定义虚函数
13     {
14         cout<<"base: "<<x<<"    "<<y<<endl;
15     }

```

```

16 };
17 class base1:public base           //定义公有派生类 base1
18 {
19 private:                         //定义派生类私有成员
20     int z;
21 public:
22     base1(int xx,int yy,int zz):base(xx,yy) //定义派生类的构造函数
23     {
24         z=zz;
25     }
26     void disp()                  //定义同名函数
27     {
28         cout<<"base1:"<<z<<endl;
29     }
30 };
31 void main()
32 {
33     base obj(3,4),*objp;         //创建基类的对象和对象指针
34     base1 obj1(1,2,5);          //创建派生类的对象
35     objp=&obj;                   //对象指针指向基类
36     objp->disp();                //调用基类成员函数
37     objp=&obj1;                  //对象指针指向派生类
38     objp->disp();                //调用派生类成员函数
39 }

```

【运行结果】在 Visual C++ 中输入上述代码，运行结果如图 12-9 所示。

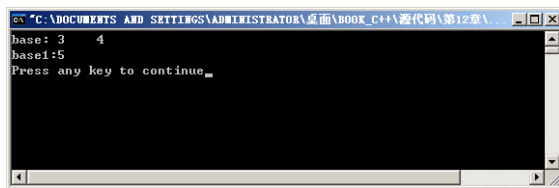


图 12-9 使用虚函数

【范例解析】同样，与上述范例 12-3 相比较，上述代码将第 12 行，基类中的 disp() 函数定义为虚函数，其他代码均不变，其返回结果就是预期要得到的正确结果。这也是由于使用虚函数实现了运行时的多态性。

提示 对于虚函数调用来说，每一个对象内部都有一个虚表指针，该虚表指针被初始化为本类的虚表。所以在程序中，不管对象类型如何转换，但该对象内部的虚表指针是固定的，所以才能实现动态的对象函数调用，这就是 C++ 多态性实现的原理。

12.3.4 多重继承和虚函数

由于多重继承可以看成是多个单继承的组合，因此多重继承的虚函数与单继承的虚函数的调用相同，一个虚函数无论被继承多少次，仍保持其虚函数的特性，与继承的次数无关。

【范例 12-6】多重继承和虚函数的应用。该范例定义了三个类，其中类 derived2 公有继承于类 derived1，类 derived1 公有继承于 base，其构成了一个多重继承。此外，在基类 base 中定义了虚函数，请读者观察，并理解它的实现，代码如代码清单 12-6 所示。

代码清单 12-6

```

1  #include <iostream.h>
2  class base           //定义基类 base

```



```

3  {
4  public:
5      virtual ~base(){};                //定义虚析构函数
6      virtual void func() const         //定义虚函数
7      {
8          cout<<"base output!"<<endl;
9      }
10 };
11 class derive1:public base              //定义派生类 derive1
12 {
13 public:
14     void func() const                  //定义成员函数
15     {
16         cout<<"derive1 output!"<<endl;
17     }
18 };
19 class derive2 :public derive1          //定义派生类 derive1 的派生类 derive2
20 {
21 public:
22     void func() const                  //定义成员函数
23     {
24         cout<<"derive2 output!"<<endl;
25     }
26 };
27 void test(const base & rBase)          //定义函数
28 {
29     rBase.func();                     //调用虚函数
30 };
31 void main()
32 {
33     base bObj;                        //创建基类 base 对象
34     derive1 d1Obj;                    //创建派生类 derive1 对象
35     derive2 d2Obj;                    //创建派生类 derive2 对象
36     test(bObj);                       //调用函数
37     test(d1Obj); //test derive1 object
38     test(d2Obj); //test derive2 object
39 }

```

【运行结果】在 Visual C++ 中输入上述代码，运行结果如图 12-10 所示。

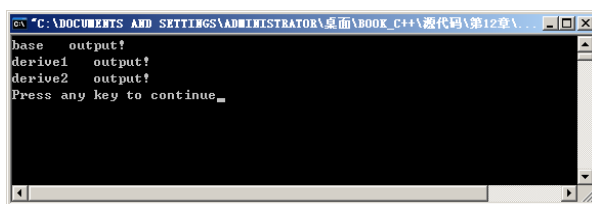


图 12-10 多重继承和虚函数的应用

【范例解析】上述代码中，定义了一个多重继承，在基类中定义了虚函数 func()，在主函数 main() 中调用该函数时，不同类创建的对象其调用的函数是不一样的，即实现了多态的“一个接口，多种实现”的功能。

注意 上述代码中在析构函数前加上关键字 virtual 进行说明，则该析构函数就被称为虚析构函数。虚析构函数的说明格式如下：

```
virtual ~<类名>()
```

在使用虚析构函数时，要注意以下几点：

- 只要基类的析构函数被声明为虚函数，则派生类的析构函数，无论是否使用 `virtual` 关键字进行声明，都自动成为虚函数。
- 如果基类的析构函数为虚函数，则当派生类未定义析构函数时，编译器所生成的析构函数也为虚函数。

一般来说，当不能决定是否应将析构函数声明为虚函数时，就将析构函数声明为虚函数。

12.3.5 虚函数的注意事项

虽然虚函数可以很好地实现多态，但在使用虚函数时应注意如下问题：

- 虚函数的声明只能出现在类函数原型的声明中，不能出现在函数体实现的时候，而且，基类中只有保护成员或公有成员才能被声明为虚函数。
- 在派生类中重新定义虚函数时，关键字 `virtual` 可以写也可以不写，但在容易引起混乱时，应写上该关键字。
- 动态联编只能通过成员函数来调用或通过指针、引用来访问虚函数，如果以对象名的形式来访问虚函数，将采用静态联编。

12.4 抽象类

抽象类是一种特殊的类，其为类提供统一的操作界面，建立抽象类的原因就是为了通过抽象类多态使用其中的成员函数，抽象类是带有纯虚函数的类。

12.4.1 纯虚函数

当在基类中不能为虚函数给出一个有意义的实现时，可以将其声明为纯虚函数。纯虚函数的实现可以留给派生类完成，纯虚函数的作用是给派生类提供一个一致的接口。一般来说，一个抽象类带有至少一个纯虚函数。纯虚函数是在一个基类中说明的虚函数，它在该基类中没有具体的操作内容，要求各派生类在重新定义时根据自己的需要定义实际的操作内容。纯虚函数的一般定义形式为：

```
virtual<函数类型><函数名>(<参数表>)=0;
```



提示 纯虚函数与普通虚函数定义的不同在于书写形式上加了“=0”，说明在基类中不用定义该函数的函数体，它的函数体由派生类定义。

【范例 12-7】纯虚函数的应用。该范例定义了类 `point`，其中定义了纯虚函数 `set()` 和 `draw()`，读者可通过下面的代码观察这两个纯虚函数的作用，代码如代码清单 12-7 所示。

代码清单 12-7

```
1  #include <iostream.h>
2  class point                                //定义基类 point
3  {
4  public:
5      point(int i=0, int j=0)                //定义构造函数
6      {
7          x0=i;
8          y0=j;
9      }
10     virtual void set() = 0;                  //声明虚函数
11     virtual void draw() = 0;                 //声明虚函数
12 protected:
```



```

13     int x0, y0;
14 };
15 class line : public point                //定义派生类 line
16 {
17 public:
18     line(int i=0, int j=0, int m=0, int n=0):point(i, j) //定义构造函数
19     {
20         x1=m;
21         y1=n;
22     }
23     void set() { cout<<"line::set() called.\n"; }      //定义成员函数
24     void draw() { cout<<"line::draw() called.\n"; }
25 protected:
26     int x1, y1;
27 };
28 class ellipse : public point             //定义派生类 ellipse
29 {
30 public:
31     ellipse(int i=0, int j=0, int p=0, int q=0):point(i, j) //定义构造函数
32     {
33         x2=p;
34         y2=q;
35     }
36     void set() { cout<<"ellipse::set() called.\n"; }    //定义成员函数
37     void draw() { cout<<"ellipse::draw() called.\n"; }
38 protected:
39     int x2, y2;
40 };
41 void drawobj(point *p)                   //定义虚函数
42 {
43     p->draw();
44 }
45 void setobj(point *p)                    //定义虚函数
46 {
47     p->set();
48 }
49 void main()
50 {
51     line *lineobj = new line;             //创建对象指针
52     ellipse *elliobj = new ellipse;
53     drawobj(lineobj);                     //调用虚函数
54     drawobj(elliobj);
55     cout<<endl;
56     setobj(lineobj);                      //调用虚函数
57     setobj(elliobj);
58     cout<<"\nRedraw the object...\n";
59     drawobj(lineobj);                     //调用虚函数
60     drawobj(elliobj);
61 }

```

【运行结果】在 Visual C++ 中输入上述代码，运行结果如图 12-11 所示。

【范例解析】上述代码中，类 line 和类 ellipse 都公有继承于类 point，在该程序中要实现多态，就必须定义主程序中要调用的 set() 函数和 draw() 函数为虚函数。同时，由于点是不需要在应用程序中画出的，因此类 point 的 set() 函数和 draw() 函数定义为纯虚函数。

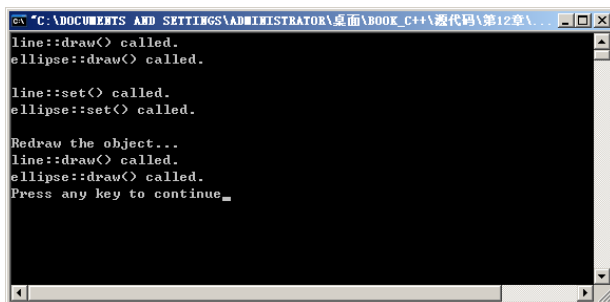


图 12-11 纯虚函数的应用

提 在主函数 main() 中新建 line 对象和 ellipse 对象后，调用这两个同名的函数即可。这就是虚函数的作用，它一般用于基类中函数没有具体操作，而派生类中该函数可能需要有操作。

12.4.2 抽象类

带有纯虚函数的类称为抽象类。抽象类是一种特殊的类，其是为了抽象和设计的目的而建立的，处于继承层次结构的较上层。抽象类是不能定义对象的，在实际中为了强调一个类是抽象类，可将该类的构造函数说明为保护的访问控制权限。

抽象类的主要作用是将有关的类组织在一个继承层次结构中，由抽象类来为它们提供一个公共的根，相关的子类是从这个根派生出来的。抽象类刻画了一组子类的操作接口的通用语义，这些语义也传给子类。一般而言，抽象类只描述这组子类共同的操作接口，而完整的实现留给子类。

抽象类只能作为基类来使用，其纯虚函数的实现由派生类给出。如果派生类没有重新定义纯虚函数，而派生类只是继承基类的纯虚函数，则这个派生类仍然还是一个抽象类。如果派生类中给出了基类纯虚函数的实现，则该派生类就不再是抽象类了，它是一个可以建立对象的具体类了。

抽象类的主要作用是给类提供统一的公共接口，以有效地发挥多态的特性。使用时应注意以下问题：

- 抽象类只能用做其他类的基类，不能建立抽象类的对象。因为它的纯虚函数没有定义功能。
- 抽象类不能用作参数类型、函数的返回类型或显式转换的类型。
- 可以声明抽象类的指针和引用，通过它们，可以并访问派生类对象，从而访问派生类的成员。
- 若抽象类的派生类中没有给出所有纯虚函数的函数体，派生类仍是一个抽象类。若抽象类的派生类中给出了所有纯虚函数的函数体，这个派生类不再是一个抽象类，它可以声明自己的对象。

【范例 12-8】抽象类的应用。该范例定义了抽象类 Vehicle，该类中包含纯虚函数 ShowMember()，该类的派生类 Car 重新定义了纯虚函数，实现其参数的输出，代码如代码清单 12-8 所示。

代码清单 12-8

```

1  #include <iostream.h>
2  class Vehicle                                //定义抽象类 Vehicle
3  {
4  public:
5      Vehicle(float speed,int total)           //定义构造函数

```




```

6      {
7          Vehicle::speed = speed;
8          Vehicle::total = total;
9      }
10     virtual void ShowMember()=0;           //定义纯虚函数
11 protected:                               //定义保护成员
12     float speed;
13     int total;
14 };
15 class Car:public Vehicle                  //定义派生类
16 {
17 public:
18     Car(int aird,float speed,int total):Vehicle(speed,total)
19                                         //定义构造函数
19     {
20         Car::aird = aird;
21     }
22     virtual void ShowMember()             //派生类成员函数重载
23     {
24         cout<<"the speed is : "<<speed<<endl;
25         cout<<"the total is : "<<total<<endl;
26         cout<<"the aird is : "<<aird<<endl;
27     }
28 protected:                               //定义保护成员
29     int aird;
30 };
31 void main()
32 {
33     //Vehicle a(100,4);                    //错误,抽象类不能创建对象
34     Car b(250,150,4);                     //创建对象
35     b.ShowMember();                       //调用成员函数
36 }
37 
```

【运行结果】在 Visual C++中输入上述代码,运行结果如图 12-12 所示。

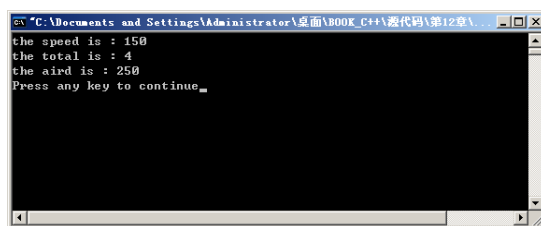


图 12-12 抽象类的应用

【范例解析】上述代码中,基类 Vehicle 定义了纯虚函数 ShowMember(),该函数并没有实际的操作意义,类 Car 公有派生于类 Vehicle,它重载了该成员函数 ShowMember()。在主函数 main()中,创建了类 Car 的对象 b,并对对象 b 进行了初始化,调用 ShowMember()函数后,其显示的是派生类 Car 的参数,因此,符合预期的结果。



警告 由于抽象类是不能创建对象的,因此上述代码的第 34 行,在主函数 main()中由类 Vehicle 来创建对象 a 的代码是错误的。

12.5 小结

本章主要介绍了 C++面向对象程序设计的又一个重要特征——多态。简单地讲,多态就是

要在程序中实现“一个接口，多种实现”，或者说多个属于不同类的函数可共用一个函数名。这和前面介绍的函数的重载不同的是：多态要求函数的参数个数、参数类型和返回类型等都完全相同。本章重点介绍了多态实现的两种形式：函数重载和虚函数。其中函数重载主要实现编译时的多态，即静态多态性，而虚函数主要实现运行时的多态，即动态多态性。最后，本章通过具体的范例就纯虚函数和抽象类的概念和应用做了简要的介绍。

12.6 习题

1. 定义一个 `Rectangle` 类，有长 `itsWidth`、宽 `itsLength` 等属性，重载其构造函数 `Rectangle()` 和 `Rectangle(int width,int length)`。

【解答】该习题主要考查重载成员函数的定义和使用。上述习题要求定义一个 `Rectangle` 类，将其构造函数进行重载。读者可以看到，该类的两个构造函数的参数是不同的，因此将其重载时要注意重载成员函数的参数须与这两个构造函数的参数一致。其简要的实现代码如下所示。

```
class Rectangle
{
public:
    Rectangle();
    Rectangle(int width, int length);
    ~Rectangle() {}
    int GetWidth() const { return itsWidth; }
    int GetLength() const { return itsLength; }
private:
    int itsWidth;
    int itsLength;
};

Rectangle::Rectangle()
{
    itsWidth = 5;
    itsLength = 10;
}

Rectangle::Rectangle (int width, int length)
{
    itsWidth = width;
    itsLength = length;
}
```

2. 定义一个哺乳动物 `Mammal` 类，再由此派生出狗 `Dog` 类，二者都定义 `Speak()` 成员函数，基类中定义为虚函数，定义一个 `Dog` 类的对象，调用 `Speak` 函数，观察运行结果。

【解答】该习题主要考查虚函数的定义和实现。在类 `Mammal` 中定义 `Speak` 成员函数为虚函数，在派生类 `Dog` 中调用该 `Speak` 函数。根据虚函数的特性，将基类中的 `Speak` 成员函数声明为虚函数后，就可以在派生类 `Dog` 中重新定义。在派生类中重新定义时，其函数原型，包括返回类型、函数名、参数个数和类型、参数的顺序都必须与基类中的原型完全一致。其简要的实现代码如下所示。

```
class Mammal
{
public:
    Mammal():itsAge(1) { cout << "Mammal constructor...\n"; }
    ~Mammal() { cout << "Mammal destructor...\n"; }
    virtual void Speak() const { cout << "Mammal speak!\n"; }
};

class Dog : public Mammal
```



```

{
public:
    Dog() { cout << "Dog Constructor...\n"; }
    ~Dog() { cout << "Dog destructor...\n"; }
    void Speak()const { cout << "Woof!\n"; }
};

int main()
{
    Mammal *pDog = new Dog;
    pDog->Speak();
}

```

3. 定义一个 Shape 抽象类, 在此基础上派生出 Rectangle 和 Circle, 两者都由 GetArea() 函数计算对象的面积, 由 GetPerim() 函数计算对象的周长。

【解答】该习题主要考查抽象类的定义和使用。该习题定义抽象类后须定义两个纯虚函数 GetArea 和 GetPerim, 用于继承于该类的派生类进行定义。其中, 在定义派生类 Rectangle 和 Circle 的同时都必须对基类的这两个纯虚函数进行重定义, 分别实现计算矩形和圆的面积和周长。其简要的实现代码如下所示。

```

class Shape
{
public:
    Shape(){}
    ~Shape(){}
    virtual float GetArea() =0 ;
    virtual float GetPerim () =0 ;
};

class Circle : public Shape
{
public:
    Circle(float radius):itsRadius(radius){}
    ~Circle(){}
    float GetArea() { return 3.14 * itsRadius * itsRadius; }
    float GetPerim () { return 6.28 * itsRadius; }
private:
    float itsRadius;
};

class Rectangle : public Shape
{
public:
    Rectangle(float len, float width): itsLength(len), itsWidth(width){};
    ~Rectangle(){};
    virtual float GetArea() { return itsLength * itsWidth; }
    float GetPerim () { return 2 * itsLength + 2 * itsWidth; }
    virtual float GetLength() { return itsLength; }
    virtual float GetWidth() { return itsWidth; }
private:
    float itsWidth;
    float itsLength;
};

```

4. 定义一个基类 BaseClass, 从它派生出类 DerivedClass, BaseClass 中定义虚析构函数, 在主程序中将一个 DerivedClass 的对象地址赋给一个 BaseClass 的指针, 观察运行过程。

【解答】该习题主要考查虚构造函数和虚析构函数的实现, 并通过指针进行操作。该习题首先必须定义基类和公有继承于基类的派生类, 在基类中定义虚析构函数。在主函数中创建派生类对象后关闭, 该析构函数将被执行。其简要的实现代码如下所示。

```

class BaseClass {
public:
    virtual ~BaseClass() {
        cout << "~BaseClass()" << endl;
    };

    class DerivedClass : public BaseClass {
    public:
        ~DerivedClass() {
            cout << "~DerivedClass()" << endl;
        };

    int main()
    {
        BaseClass* bp = new DerivedClass;
        delete bp;
    }
}

```

5. 使用抽象类和纯虚函数来访问数据结构中的队列和堆栈，队列是一个先进先出的数据结构，而堆栈是一个后进先出的数据结构。

【解答】上述程序可首先定义基类 `list` 为一个抽象类，其中定义了向单链表中保存值的纯虚函数 `store()` 和从中读取值的纯虚函数 `retrieve()`。抽象类 `list` 派生了两个队列类 `queue` 和堆栈类 `stack`，它们根据各自的需要，重新定义纯虚函数 `store()` 和 `retrieve()`。其简要的实现代码如下所示。

```

class list                                     //定义基类
{
public:                                         //定义公有成员
    list *head;                               //定义类类型的指针
    list *tail;                               //定义类类型的指针
    list *next;                               //定义类类型的指针
    int num;                                  //定义整型变量
    list()                                    //基类的构造函数
    {
        head=tail=next=NULL;                //头、尾和 next 指针均为 NULL
    }
    virtual void store(int i)=0;              //声明纯虚函数
    virtual int retrieve()=0;                 //声明虚函数
};

class queue:public list                       //定义派生类队列类
{
public:                                       //定义公有成员
    void store(int i);                      //声明进入队列成员函数
    int retrieve();                          //声明出队列成员函数
};

class stack:public list                      //定义派生类栈
{
public:                                     //定义公有成员
    void store(int i);                      //声明公有成员函数
    int retrieve();                          //声明公有成员函数
};

```

第 13 章 运算符重载

第 12 章介绍了多态的实现技术，主要包括函数的重载和虚函数。事实上，本章介绍的运算符重载也是多态性实现的一个重要手段。运算符重载实现的是编译时的多态，即静态多态性。C++预定义的运算符只是对基本数据类型进行操作，而对于自定义的数据类型比如类，却没有类似的操作。为了实现对自定义类型的操作，就必须自己编写程序来说明某个运算符作用在这些数据类型上时，应该完成怎样的操作，这就要引入运算符重载的概念。

以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为一个学习的参照。

- 理解运算符重载的概念及定义。
- 掌握运算符重载的两种形式及其实现。
- 掌握特殊运算符的重载。

13.1 运算符重载简介

简单地说，运算符重载是实现编译时多态性的另外一种形式。运算符重载是对已有的运算符赋予多重含义，使同一个运算符作用于不同类型的数据时，实现不同类型的行为。

13.1.1 运算符重载的定义

C++提供了许多库函数，如字符串操作库函数、数学运算库函数和图形操作库函数等。同时也提供了许多标准运算符，如+、-、*、/、&等。这些运算符就像 C++的库函数一样，可以实现一定的功能，是程序开发的基本工具。

关于函数的重载，通过前面章节的学习，读者已经知道能够对函数进行重载。同样，对于运算符，C++也提供了重载的机会。运算符重载就是运用函数重载的方法，对 C++提供的标准运算符重新定义，以完成某种特定的操作。

运算符重载的实质是函数重载。事实上，C++语言中的每一个运算符对应着一个运算符函数，在实现过程中，把指定的运算表达式中的运算符转化为对运算符函数的调用，而表达式中的运算对象转化为运算符函数的实参，这个过程是在编译阶段完成的。例如：

```
int a=1,b=2;  
a+b;
```

表达式“a+b”在编译前，将被解释为函数调用形式：operator+(a,b)。其中，operator 是一个关键字，它与后面的“+”共同组成了该运算符函数的函数名。因此，可以将运算符重载看作是一种特殊的函数重载。



提示 运算符重载是通过创建运算符函数实现的，运算符函数定义了重载的运算符将要进行的操作。运算符函数的定义与其他函数的定义类似，唯一的区别是运算符函数的函数名是由关键字 operator 和其后要重载的运算符符号构成的。

运算符函数定义的一般格式如下：

```
<返回类型说明符> operator <运算符符号>(<参数表>)  
{  
    <函数体>  
}
```



运算符重载的目的是设置 C++ 语言中的某一运算符,让它们之间并不冲突,C++ 语言会根据运算符的位置辨别应使用哪一种功能进行运算。可见,运算符重载的优点是允许改变使用于系统内部的运算符的操作方式,以适应用户新定义类型的类似运算。

13.1.2 运算符重载的特点

尽管运算符重载是一种特殊的函数重载,但相比函数重载,运算符重载有着自身的一些特点。使用 `operator` 关键字对重载函数进行标识和定义。运算符有三种形式,即中缀、后缀、前缀,它们的 `operator` 表示形式如表 13-1 所示。

表 13-1 三种运算符的 `operator` 表示形式

运算符分类	常规表示	<code>operator</code> 表示形式	参数个数
中缀	<code>a+b</code>	<code>operator +(a,b)</code>	二元
前缀	<code>-a</code>	<code>operator -(a,0)</code>	一元
后缀	<code>a++</code>	<code>operator ++(a)</code>	一元

(1) 只能重载 C++ 提供的标准运算符。可重载的运算符在表 13-2 中列出。

表 13-2 可重载的 C++ 运算符

<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>	<code>^</code>	<code>&</code>	<code> </code>	<code>~</code>	<code>!</code>	<code>=</code>	<code><></code>	<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>	<code>%=</code>	<code>^=</code>	<code>&=</code>	<code> =</code>
<code><<</code>	<code>>></code>	<code><<=</code>	<code>>>=</code>	<code>=</code>	<code>=</code>	<code><</code>	<code>></code>	<code>&&</code>	<code> </code>	<code><<</code>	<code>>></code>	<code><<=</code>	<code>>>=</code>	<code>=</code>	<code>=</code>	<code><</code>	<code>></code>	<code>&&</code>	<code> </code>
<code>++</code>	<code>-</code>	<code>{}</code>	<code>()</code>	<code>-></code>	<code>,</code>	<code>new</code>	<code>new[]</code>	<code>delete</code>	<code>delete[]</code>	类型转换运算符									



类型转换运算符包括 `int`、`char`、`long`、`float`、`int*`、`char*` 等。不能重载的运算符是：“.”、
注意 “*”、“::”、“sizeof” 和 “?:”。

(2) 参数个数固定。重载函数的参数个数与标准运算符保持一致。即对于一元运算符(前缀和后缀形式),重载函数有且只能有一个参数;对于二元运算符(中缀形式),重载函数有且只能有两个参数。

(3) 针对类对象进行操作。即重载函数的参数至少有一个属于 `class` 类型。

根据上述运算符重载的定义及其特点,下面通过一个范例来理解在实际的程序中是如何进行运算符重载的。

【范例 13-1】重载 “+” 运算符。该范例对 C++ 的标准算术运算符 “+” 进行重载,使得 “+” 运算符可以进行类与整数之间的算术运算,实现代码如代码清单 13-1 所示。

代码清单 13-1

```
1  #include <iostream.h>
2  class CAdd                                //定义类 CAdd
3  {
4  public:
5      int m_Operand;
6      CAdd()                                //定义构造函数
7      {
8          m_Operand=0;
9      }
10     CAdd(int value)                        //重载构造函数
11     {
12         m_Operand=value;
13     }
14 };;
```



```

15  CAdd operator +(CAdd a, int b)           //重载“+”运算符，操作 CAdd 类
16                                           //第一个参数类型是 CAdd，返回类型是 CAdd
17  {
18      CAdd sum;                           //创建对象
19      sum.m_Operand=a. m_Operand +b;       //实现类中成员变量与指定整型变量相加功能
20      return sum;
21  }
22  void main()
23  {
24      CAdd a(5),b;                         //创建对象
25      b=a+8;                               //调用重载后的“+”运算符
26      cout<<"the sum is: "<<b.m_Operand<<endl; //输出
27  }

```

【运行结果】在 Visual C++中新建一个【C++ Source File】文件，输入如上的代码，编译无误后运行，其结果如图 13-1 所示。

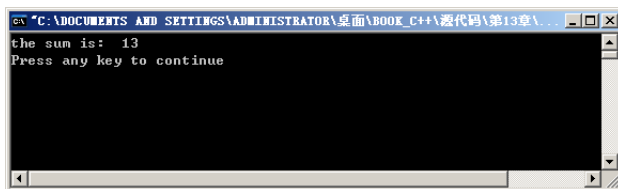


图 13-1 重载+运算符

【范例解析】上述代码中，定义了类 CAdd，使用运算符重载的定义形式重载了“+”运算符，该运算符实现类的成员变量与整型变量进行算术相加运算，返回 CAdd 类型的值。在主函数 main()中创建了两个类 CAdd 的对象 a、b，使用重载后的“+”运算符实现运算“b=a+8”。



提示 读者可以看出，重载后的运算符与 C++的标准算术运算符的使用形式是一样的，这样极大地方便了程序设计人员编程。

13.1.3 运算符重载的规则

根据前面关于运算符的定义和特点，读者可以看出运算符重载时实现的方法是：先把指定的运算表达式转化为对运算符函数的调用，运算对象转化为运算符函数的实参，再根据实参的类型来确定需要调用的函数。这个过程在编译时完成，因此，可以为同一个运算符定义几个运算符重载函数来进行不同的操作。

简单地说，实现运算符重载即编写一个函数，该函数以“operator 运算符”为函数名，其定义了重载的运算符将要执行的操作。此外，函数的形参类型必须是自定义的类型。当使用该运算符对形参规定的数据类型进行运算时，就执行函数体中的操作，而覆盖了原运算符的功能。

我们知道，C++中的运算符具有特定的语法规则，因此运算符重载时也要遵守一定的规则，其主要的规则如下：

- C++中的运算符除了几个不能重载外，其他的都能重载，而且只能重载已有的运算符，不能自己另编。
- 重载以后运算符的优先级和结合性都不能改变，语法结构也不能改变，即单目运算符只能重载为单目运算符，多目运算符只能重载为多目运算符。
- 运算符重载以后的功能应与原有功能类似，含义必须清楚，不能有二义性。

13.2 运算符重载的形式

运算符的重载形式有两种：一种是重载为类的成员函数，一种是重载为类的友元函数。对于每一种重载形式，由于运算符的不同，都可以将其主要分为双目运算符和单目运算符的实现。本节将详细讲解这两种运算符的不同实现形式。

13.2.1 重载为类的成员函数

将运算符重载为它将要操作的类的成员函数，称为成员运算符函数。实际使用时，总是通过该类的某个对象访问重载的运算符。一般来说，成员运算符函数在类内进行声明，在类外进行定义。成员运算符在类内声明的一般形式为：

```
<返回类型> operator<运算符>(参数表);
```

在类外定义的一般形式为：

```
<返回类型> <类名::> operator<运算符>(参数表)
{
    函数体
}
```

其中，`operator` 是定义运算符重载函数的关键字；运算符是要重载的运算符的名称；参数表给出重载运算符所需要的参数和类型。



注意 如果将运算符重载为成员函数，则该成员函数一般只在类中声明，而将具体的定义放在类外，因此需要使用作用域运算符来实现。

13.2.2 双目运算符重载为成员函数

双目运算符重载为成员函数时，左操作数是访问该重载运算符的对象本身的数据，由 `this` 指针指出，右操作数通过成员运算符函数的参数指出。所以，此时成员运算符函数只有一个参数。如：

```
class X
{
    //
    int operator+(X ob);
}
```

双目运算符重载为成员函数后，就可以在主函数或其他类中进行调用了。C++中，一般有显式和隐式两种调用方法。

- 显式调用：对象名.operator 运算符号(参数);如 `aa.opreator+(bb);`。
- 隐式调用：对象名 重载的运算符号 对象名;如 `aa+bb;`。

【范例 13-2】双目运算符重载为成员函数。该范例重载了一个双目运算符“+”为类的成员函数，在主函数中调用该运算符，读者可以通过该范例理解其与标准的 C++ 算术运算符的区别，其实现代码如代码清单 13-2 所示。

代码清单 13-2

```
1  #include<iostream.h>
2  class point                                //定义类 point
3  {
4  private:
5      int x,y;                                //定义私有成员变量
6  public:
7      point(int xx=0,int yy=0)                //定义构造函数
```




```

8      {
9          x=xx;
10         y=yy;
11     }
12     int getx()                //定义成员函数
13     {
14         return x;
15     }
16     int gety()
17     {
18         return y;
19     }
20     point operator+(point p);  //声明运算符重载为成员函数
21 };
22 point point::operator+(point p) //定义该成员函数
23 {
24     point temp;
25     temp.x=x+p.x;            //实现成员变量之间的加法
26     temp.y=y+p.y;
27     return temp;            //返回 point 类型值
28 }
29 void main()
30 {
31     point ob1(1,2),ob2(3,4),ob3,ob4; //创建对象
32     ob3=ob1+ob2;                  //隐式调用运算符
33     ob4=ob1.operator+(ob2);       //显式调用运算符
34     cout<<"ob3.x="<<ob3.getx()<<"    ob3.y="<<ob3.gety()<<endl;
35     cout<<"ob4.x="<<ob4.getx()<<"    ob4.y="<<ob4.gety()<<endl;
36 }

```

【运行结果】在 Visual C++ 中执行上述代码，其运行结果如图 13-2 所示。

【范例解析】上述代码中，定义了类 `point`，其中第 20 行代码声明了重载双目运算符“+”为成员函数，在类外的第 22~28 行对该成员函数进行了定义，它实现类的成员变量之间的加法，并返回一个 `point` 类型的值。在主函数 `main()` 中，采用了隐式调用和显式调用两种方式实现两个类 `point` 的对象 `ob1` 和 `ob2` 之间的加法。

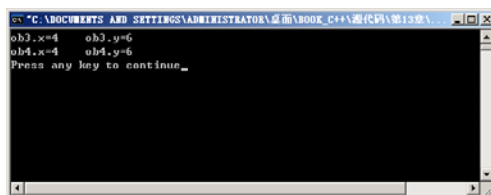


图 13-2 双目运算符重载为成员函数



提示 隐式调用和显式调用重载运算符得到的结果是一样的，在实际的程序中，隐式调用更符合人们的使用习惯。

13.2.3 单目运算符重载为成员函数

上述范例针对的是常见的双目运算符，事实上，单目运算符也可以进行重载。单目运算符重载为成员函数时，操作数是访问该重载运算符的对象本身的数据，由 `this` 指针指出，所以，此时成员运算符函数没有参数。如：

```

class X
{
    int operator++();
}

```

与双目运算符的重载类似，单目运算符重载为成员函数后，在调用时也有显式和隐式两种调用方法。

- 显式调用：对象名.operator 运算符();，如 aa.opreator++();。
- 隐式调用：重载的运算符 对象名;，如 ++aa;。

【范例 13-3】单目运算符重载为成员函数。该范例重载了一个单目运算符“++”，该运算符实现类的所有成员变量的递加操作，读者可通过该范例理解其与普通递增运算符的区别，实现代码如代码清单 13-3 所示。

代码清单 13-3

```

1  #include<iostream.h>
2  class point                                //定义类 point
3  {
4  private:
5      int x,y;                                //定义私有成员变量
6  public:
7      point(int xx=0,int yy=0)                //定义构造函数
8      {
9          x=xx;
10         y=yy;
11     }
12     int getx()                                //定义成员函数
13     {
14         return x;
15     }
16     int gety()
17     {
18         return y;
19     }
20     point operator++();                        //声明单目运算符重载为成员函数
21 };
22 point point:: operator++()                    //定义该成员函数
23 {
24     ++ x;                                    //实现递加功能
25     ++y;
26     return *this;                            //返回对象本身
27 }
28 void main()
29 {
30     point ob(3,4);                            //创建对象
31     cout<<"ob.x="<<ob.getx()<<"    ob.y="<<ob.gety()<<endl;
32     ++ob;                                    //隐式调用运算符
33     cout<<"ob.x="<<ob.getx()<<"    ob.y="<<ob.gety()<<endl;
34     ob.operator++();                            //显式调用运算符
35     cout<<"ob.x="<<ob.getx()<<"    ob.y="<<ob.gety()<<endl;
36 }

```

【运行结果】在 Visual C++中执行上述代码，其运行结果如图 13-3 所示。

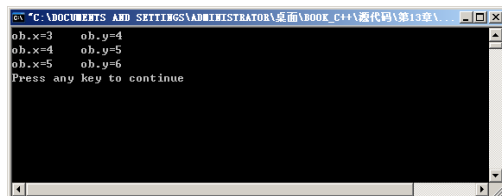


图 13-3 单目运算符重载为成员函数

【范例解析】上述代码中定义了类 point，其中第 20 行代码声明了重载单目运算符“++”为成员函数，在类外的第 22~27 行对该成员函数进行了定义，其实现类的成员变量的递加，



返回一个 `point` 类型的值。在主函数 `main()` 中, 采用了隐式调用和显式调用两种方式实现类 `point` 的对象 `ob` 中的成员变量 `x` 和 `y` 的递加。



注意 在定义单目运算符重载的成员函数时, 其返回值用 `this` 指针指出。有关单目运算符后缀方式的重载将在后面章节介绍。

13.2.4 运算符重载为类的友元函数

将重载的运算符函数定义为类的友元函数, 称为友元运算符函数。友元运算符函数不是类的成员, 它在类内声明原型, 在类外定义函数本身。由于友元运算符函数不是类的成员函数, 不属于任何一个类对象, 所以没有 `this` 指针。因此, 重载双目运算符时要有两个参数, 重载单目运算符时只要一个参数就可以了。一般来说, 友元运算符函数在类内声明的一般形式为:

```
friend<返回类型> operator<运算符>(参数表);
```

在类外定义的一般形式为:

```
<返回类型> operator<运算符>(参数表)
{
    函数体
}
```

其中, 参数说明如下。

- **friend:** 表示声明友元函数的关键字。
- **operator:** 是定义运算符重载函数的关键字。
- **运算符:** 是要重载的运算符的名称。
- **参数表:** 给出重载运算符所需要的参数和类型。



提示 如果将运算符重载为友元函数, 该函数同样只在类中声明为友元函数, 而具体的定义也放在类外。

13.2.5 双目运算符重载为友元函数

双目运算符重载为友元函数时, 由于没有 `this` 指针, 所以两个操作数都要通过友元运算符函数的参数指出。与运算符重载为类的成员函数类似, 双目运算符重载为友元函数后, 其调用也有显式和隐式两种方法。

- 显式调用: `operator 运算符号 (参数 1, 参数 2);`, 如 `opreator+(aa,bb);`。
- 隐式调用: 对象名 重载的运算符号 对象名;, 如 `aa+bb;`。

【范例 13-4】 双目运算符重载为友元函数。该范例在重载了双目运算符 “+” 为类的友元函数, 其实现类的两个成员变量之间的算术和, 读者可将其与重载运算符为成员函数的作比较, 代码如代码清单 13-4 所示。

代码清单 13-4

```
1  #include<iostream.h>
2  class point                                //定义类point
3  {
4  private:
5      int x,y;                                //定义私有成员变量
6  public:
7      point(int xx=0,int yy=0)                //定义构造函数
```

```

8      {
9          x=xx;
10         y=yy;
11     }
12     int getx()                //定义成员函数
13     {
14         return x;
15     }
16     int gety()                //定义成员函数
17     {
18         return y;
19     }
20     friend point operator+(point p,point q);    //声明运算符重载为友元函数
21 };
22 point operator+(point p,point q)              //定义该友元函数
23 {
24     point temp;
25     temp.x=p.x+q.x;                          //实现成员变量之间的加法
26     temp.y=p.y+q.y;
27     return temp;                             //返回 point 类型值
28 }
29 void main()
30 {
31     point ob1(1,2),ob2(3,4),ob3,ob4;          //创建对象
32     ob3=ob1+ob2;                              //隐式调用运算符
33     ob4=operator+(ob1,ob2);                   //显式调用运算符
34     cout<<"ob3.x="<<ob3.getx()<<"    ob3.y="<<ob3.gety()<<endl;
35     cout<<"ob4.x="<<ob4.getx()<<"    ob4.y="<<ob4.gety()<<endl;
36 }

```

【运行结果】在 Visual C++ 中执行上述代码，其运行结果如图 13-4 所示。

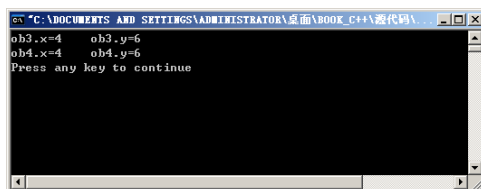


图 13-4 双目运算符重载为友元函数

【范例解析】上述代码中，定义了类 `point`，其中第 20 行代码声明了重载双目运算符“+”为友元函数，在类外的第 22~28 行对该友元函数进行了定义，它实现类的成员变量之间的加法，并返回一个 `point` 类型的值。在主函数 `main()` 中，采用了隐式调用和显式调用两种方式实现两个类 `point` 的对象 `ob1` 和 `ob2` 之间的加法。

注 在类外定义友元函数不需要加类名和作用域运算符，因为友元函数并不属于某一个类。此外，读者也可看出，双目运算符重载为友元函数和双目运算符重载为成员函数的根本区别在于其操作数的个数不同，前者定义时只需一个，而后者需要指定两个参数。

13.2.6 单目运算符重载为友元函数

与单目运算符重载为成员函数相同，单目运算符重载为友元函数时，由于没有 `this` 指针，所以操作数要通过友元运算符函数的参数指出。同样，单目运算符重载为友元函数后也有显式和隐式两种调用方法。



- 显式调用：operator 运算符号（参数）；，如 opreator++(aa);。
- 隐式调用：对象名 重载的运算符号 对象名；，如++aa;。

【范例 13-5】单目运算符重载为友元函数。该范例重载了一个单目运算符“++”为友元函数，该运算符实现类的所有成员变量的递加操作，读者可通过该范例理解其与重载为成员函数的运算符的区别，其代码如代码清单 13-5 所示。

代码清单 13-5

```

1  #include<iostream.h>
2  class point                                //定义类point
3  {
4  private:
5      int x,y;                                //定义私有成员变量
6  public:
7      point(int xx=0,int yy=0)                //定义构造函数
8      {
9          x=xx;
10         y=yy;
11     }
12     int getx()                                //定义成员函数
13     {
14         return x;
15     }
16     int gety()                                //定义成员函数
17     {
18         return y;
19     }
20     friend point operator++(point &p);        //声明单目运算符重载为友元函数
21 };
22 point operator++(point &p)                    //定义该友元函数
23 {
24     ++ p.x;                                    //实现递加功能
25     ++p.y;
26     return p;                                    //返回对象类型
27 }
28 void main()
29 {
30     point ob(3,4);                                //创建对象
31     cout<<"ob.x="<<ob.getx()<<"    ob.y="<<ob.gety()<<endl;
32     ++ob;                                    //隐式调用运算符
33     cout<<"ob.x="<<ob.getx()<<"    ob.y="<<ob.gety()<<endl;
34     operator++(ob);                            //显式调用运算符
35     cout<<"ob.x="<<ob.getx()<<"    ob.y="<<ob.gety()<<endl;
36 }
```

【运行结果】在 Visual C++中执行上述代码，其运行结果如图 13-5 所示。

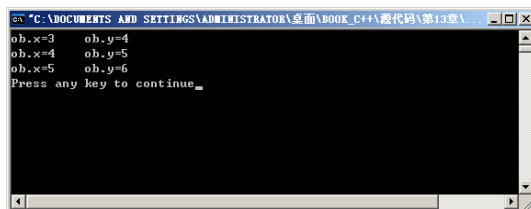


图 13-5 单目运算符重载为友元函数

【范例解析】上述代码中定义了类 point，其中第 20 行代码声明了重载单目运算符“++”为友元函数，在类外的第 22~27 行对该成员函数进行了定义，它实现类的成员变量的递加，

并返回一个 `point` 类型的值。在主函数 `main()` 中, 采用了隐式调用和显式调用两种方式实现类 `point` 的对象 `ob` 中的成员变量 `x` 和 `y` 的递增。

提示 在将运算符重载为友元函数时, 读者需要注意, 除运算符 `:=`、`()`、`[]`、`->` 不能用友元函数重载外, 其余的运算符都可以进行重载。此外, 使用友元函数重载单目运算符 “++” 和 “--” 时, 由于要改变操作数自身的值, 所以应采用引用参数传递操作数, 否则会出现错误。

13.2.7 成员运算符函数与友元运算符函数的比较

前面介绍了将运算符重载为成员函数和将运算符重载为友元函数, 每一种都分别通过范例来讲解其应用。现在对这两种方法进行比较, 其结果如下。

- 对双目运算符: 成员运算符函数是类的成员, 带有 `this` 指针, 只需一个参数, 而友元运算符函数不是类的成员, 不带 `this` 指针, 参数必须是两个。对单目运算符: 成员运算符函数不带参数, 而友元运算符必须带一个参数。
- 双目运算符可被重载为友元函数, 也可被重载为成员函数, 但在运算符的左操作数是一个标准数据类型而右操作数是对象的情况下, 必须将它重载为友元函数, 原因是标准数据类型的数据不能产生对重载运算符的调用。
- 成员运算符函数与友元运算符函数都可用习惯和专用方式调用。

总的来说, 将运算符重载为成员函数还是将运算符重载为友元函数, 选择哪种方式比较好, 要根据实际情况和使用习惯决定。一般而言, 对于双目运算符重载为友元运算符函数较好, 若运算符的操作数特别是左操作数需要进行类型转换, 必须重载为友元运算符函数。若一个运算符需要修改对象的状态, 则选择成员运算符较好。

13.3 特殊运算符的重载

读者可以看出, 前面介绍运算符重载的范例, 基本都只涉及简单的算术运算符的重载, 本节将介绍几种特殊运算符的重载。

13.3.1 “++” 和 “--” 的重载

前面范例中介绍了 “++” 和 “--” 的重载, 但运算符 “++” 和 “--” 有前置和后置两种形式。例如, 表达式 “`a++`” 和表达式 “`++a`” 是不一样的。

如果不区分前置和后置, 则使用 `operator++()` 或 `operator--()` 即可; 否则, 要使用 `operator++()` 或 `operator--()` 来重载前置运算符, 使用 `operator++(int)` 或 `operator--(int)` 来重载后置运算符, 调用时, 参数 `int` 被传递给值 0。读者可以看出, 前面介绍的 “++” 和 “--” 的重载都是前置的形式, 而本节将要介绍的是运算符 “++” 和 “--” 重载的后置形式。

【范例 13-6】 “++” 和 “--” 运算符的重载。范例实现了运算符 “++” 和 “--” 的重载, 包括其前置和后置两种形式, 读者可以观察其定义和调用的区别, 代码如代码清单 13-6 所示。

代码清单 13-6

```
1  #include<iostream.h>
2  class point                      //定义类point
3  {
4  private:
5      int x,y;
```



```

6   public:
7       point(int xx=0,int yy=0)           //定义构造函数
8       {
9           x=xx;
10          y=yy;
11      }
12      int getx()                         //定义成员函数
13      {
14          return x;
15      }
16      int gety()                         //定义成员函数
17      {
18          return y;
19      }
20      point operator++();                //声明前置运算符重载
21      point operator--(int);             //声明后置运算符重载
22  };
23  point point::operator++()              //定义前置运算符重载
24  {
25      ++x;
26      ++y;
27      return *this;                     //返回类类型的值
28  }
29  point point::operator--(int)           //定义后置运算符重载
30  {
31      x--;
32      y--;
33      return *this;                     //返回类类型的值
34  }
35  void main()
36  {
37      point ob(3,4);                     //创建对象
38      cout<<"ob.x="<<ob.getx()<<"    ob.y="<<ob.gety()<<endl;
39      ++ob;                              //调用前置运算符++
40      cout<<"ob.x="<<ob.getx()<<"    ob.y="<<ob.gety()<<endl;
41      ob.operator--(0);                   //调用后置运算符--
42      cout<<"ob.x="<<ob.getx()<<"    ob.y="<<ob.gety()<<endl;
43  }

```

【运行结果】在 Visual C++ 中执行上述代码，其运行结果如图 13-6 所示。

【范例解析】上述代码中，将运算符“++”和“--”重载为类的成员函数。其中，运算符“++”采用前置方式重载，在主函数 main() 中调用时采用了隐式调用。而运算符“--”采用后置方式重载，在主函数 main() 中调用时采用了显式调用。

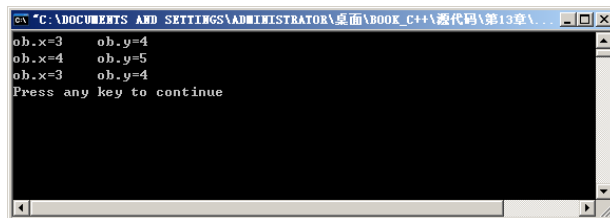


图 13-6 “++”和“--”运算符的重载。



注意 从以上代码中读者可以看出，声明和定义后置的“++”或“--”等运算符重载，都必须含有形式参数，在调用时一般为它指定实参 0。

13.3.2 赋值运算符“=”的重载

除了前面讲解的算术运算符可以进行重载外，许多其他类型的运算符也可以进行重载，赋值运算符“=”也是如此。在进行赋值运算符“=”重载的应用前，读者应先了解赋值运算符“=”在实际程序中的运行情况。事实上，对于任何一个类，如果没有用户自定义的赋值运算符函数，系统会自动地为其生成一个默认的赋值运算符函数，以完成数据成员之间的复制。例如，下面的程序段：

```
X & X::operator=(const X &source)
{
    //类对象成员之间的赋值语句
}
```

一旦类 X 的两个对象 ob1 和 ob2 已创建，就可用 ob1=ob2 进行赋值了。通常情况下，默认的赋值运算符函数就可完成赋值任务，但在某些特殊情况下。例如，类中有一种指针类的形式，如果使用默认的赋值运算符函数就会产生指针悬挂的错误。此时，就必须显式地定义一个赋值运算符重载函数，使参与赋值的两个对象有各自的存储空间，以解决这个问题。

【范例 13-7】赋值运算符“=”的重载。范例实现了赋值运算符“=”的重载，重载后的赋值运算符能够实现网站名称和地址等字符串之间的赋值，代码如代码清单 13-7 所示。

代码清单 13-7

```
1  # include <iostream.h>
2  # include <string.h>                                //声明头文件
3  class Internet                                       //定义类 Internet
4  {
5  public:
6      Internet(char *name,char *url)                  //定义构造函数
7      {
8          Internet::name = new char[strlen(name)+1];
9          Internet::url = new char[strlen(url)+1];
10         if(name)                                     //name 不为 0
11         {
12             strcpy(Internet::name,name);             //字符串复制
13         }
14         if(url)                                       //url 不为 0
15         {
16             strcpy(Internet::url,url);
17         }
18     }
19     Internet(Internet &temp)                          //拷贝构造函数
20     {
21         Internet::name=new char[strlen(temp.name)+1]; //申请空间
22         Internet::url=new char[strlen(temp.url)+1];
23         if(name)                                       //空间申请成功
24         {
25             strcpy(Internet::name,temp.name);        //字符串复制
26         }
27         if(url)
28         {
29             strcpy(Internet::url,temp.url);
30         }
31     }
32     ~Internet()                                       //定义析构函数
33     {
34         delete[] name;                                //释放空间
35         delete[] url;
```




```

36     }
37     Internet& operator =(Internet &temp)           //赋值运算符重载函数
38     {
39         delete[] this->name;
40         delete[] this->url;
41         this->name = new char[strlen(temp.name)+1]; //分配新空间
42         this->url = new char[strlen(temp.url)+1];
43         if(this->name)
44         {
45             strcpy(this->name,temp.name);           //字符串复制
46         }
47         if(this->url)
48         {
49             strcpy(this->url,temp.url);             //字符串复制
50         }
51         return *this;                               //返回类类型
52     }
53 public:
54     char *name;                                     //定义公有成员变量
55     char *url;
56 };
57 void main()
58 {
59     Internet a("Education","www.edu.cn");          //创建对象
60     Internet b = a;                                //b 对象还不存在, 所以调用拷贝
                                                    //构造函数, 进行构造处理
61     cout<<b.name<<endl<<b.url<<endl;
62     Internet c("Tsinghua","www.tsinghua.edu.cn"); //创建对象
63     b = c;                                          //b 对象已经存在, 所以系统选择
                                                    //赋值运算符重载函数处理
64     cout<<b.name<<endl<<b.url<<endl;
65 }

```

【运行结果】在 Visual C++ 中执行上述代码, 其运行结果如图 13-7 所示。

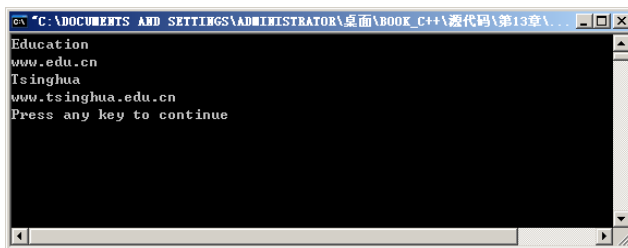


图 13-7 赋值运算符“=”的重载

【范例解析】上述代码中, 第 37 行的语句“Internet& operator =(Internet &temp)”就是赋值运算符重载函数的定义, 其功能定义在第 37~51 行。其中, 由于 b 对象已经构造过, name 和 url 指针的范围已经确定, 所以在复制新内容进去之前必须把存储区清除。由于区域的过大和过小都不好, 所以跟在后面重新分配堆区大小, 而后进行复制工作。

在类对象还未存在的情况下, 赋值过程通过拷贝构造函数进行构造处理 (代码中的 Internet b = a; 就是这种情况), 但当对象已经存在, 那么赋值过程就通过赋值运算符重载函数处理 (例子中的 b = c; 就属于此种情况)。



注意 在进行赋值运算符“=”的重载时, 读者需要注意的是, 赋值运算符只能重载为成员运算符函数, 不能重载为友元运算符函数。此外, 赋值运算符重载后不能被继承。

13.3.3 下标运算符“[]”的重载

下标运算符 `operator[]` 通常用来访问数组中的某个元素。事实上，其可以看作一个双目运算符，第一个运算符是数组名，第二个运算符是数组下标。在类对象中，可以重载下标运算符，用它来定义相应对象的下标运算。一般来说，下标运算符定义的形式如下：

```
T1 T::operator[](T2);
```

其中，参数说明如下。

- **T**：是定义下标运算符的类，其不必是常量。
- **T2**：表示下标，其可以是任意类型，如整型、字符型或某个类。
- **T1**：是数组运算的结果，其也可以是任意类型，但为了能对数组赋值，一般将其声明为引用形式。

在有了上面的定义之后，在实际的程序中，可以通过下面两种方法来调用下标运算符（`x` 为数组名或对象，`y` 为下标）：

```
x[y]
```

或

```
x.operator[](y)
```

【范例 13-8】下标运算符“[]”的重载。该范例实现了下标运算符“[]”的重载，重载后的下标运算符能够判断数组的下标是否越界，如越界则给出错误信息，实现代码如代码清单 13-8 所示。

代码清单 13-8

```

1  # include <iostream.h>
2  # include <stdlib.h>
3  class ainteger                                //定义类 ainteger
4  {
5      int *a;                                    //定义私有成员
6      int sz;
7  public:
8      ainteger(int size)                        //定义构造函数
9      {
10         sz=size;
11         a=new int[size];
12     }
13     int &operator[](int i)                    //重载下标运算符[]
14     {
15         if(i<0||i>=sz)                        //判断是否越界
16         {
17             cout<<"error"<<endl;
18             exit(1);                            //越界则异常，退出程序
19         }
20         return a[i];
21     }
22     ~ainteger()                                //定义析构函数
23     {
24         delete[]a;                            //清除地址
25     }
26 };
27 void main()
28 {
29     ainteger a(5);                            //创建对象
30     a[3]=0;                                    //赋值，不越界

```



```

31     cout<<"a[3]= "<<a[3]<<endl;
32     a.operator[](3)=0;                                //赋值, 不越界
33     cout<<"a.operator[](3)= "<<a[3]<<endl;
34     cout<<"a[6]= ";                                    //赋值, 越界
34     a.operator[](6)=6;
35 }

```

【运行结果】在 Visual C++中执行上述代码, 其运行结果如图 13-8 所示。

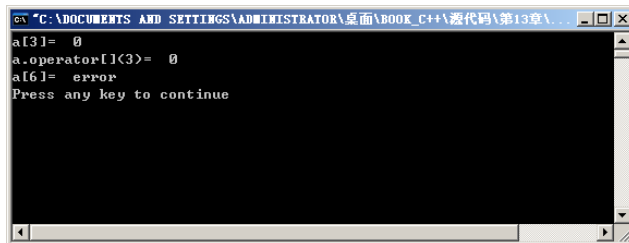


图 13-8 下标运算符“[]”的重载

【范例解析】上述代码中, 第 13~21 行实现了下标运算符“[]”的重载, 其增加了判断数组下标是否越界的功能, 下标越界则返回“error”的错误信息, 否则返回正确的数组元素值。在主函数 main()中创建一个对象 a, 其分别采用上述两种方式调用该下标运算符, 如未越界则返回数组元素的值, 否则返回“error”错误信息。



警告 C++不允许把下标运算符函数作为外部函数来定义, 其只能是非静态的成员函数。

13.4 类类型转换

类类型是指某个对象的数据类型为类, 而不是标准的数据类型。在 C++中, 标准的数据类型与类类型之间的转换有三种方法。

- 通过构造函数转换: 通过构造函数能将标准数据类型向类类型转换, 但不能将类类型转换为标准类型。
- 通过类类型转换函数转换: 要将类类型转换为标准数据类型时, 需要采用显式类型转换机制, 定义类类型转换函数。
- 通过运算符重载实现类型转换: 可以实现标准类型的数据与类对象之间的运算。

其中, 通过类类型转换函数转换需要定义一个类的类型转换函数。一般来说, C++中定义一个类的类型转换函数的形式为:

```

<类名>::operator type()
{
    return type 类型的数据    //返回 type 类型的对象
}

```

其中, type 为要专项的目标类型, 通常是标准类型。但是, 使用类类型转换函数进行类型转换时, 需要注意的是:

- 此函数的功能是将类的对象转换为类型为 type 的数据, 它既没有参数, 也没有返回类型, 但在函数体中必须返回具有 type 类型的一个数据。
- 类类型转换函数只能定义为类的成员函数, 可在类内声明类外定义, 也可在类内定义。
- 一个类内可定义多个类类型转换函数, 编译器会根据操作数的类型自动选择一个合适的类型转换函数与之匹配。但在可能出现二义性时, 应显式地使用类类型转换函数进行转换。



在实际的程序中,使用最多的还是通过运算符重载来实现类型的转换,转换运算符的声明方式比较特别,其声明方法如下:

```
perator 类名();
```

【范例 13-9】运算符重载转换运算符。范例在类中重载了一个类型转换函数 `int()`,该函数实现输出并返回类类型值,读者可仔细理解下面的代码,如代码清单 13-9 所示。

代码清单 13-9

```
1  #include <iostream.h>
2  class Test                                //定义类
3  {
4  public:
5      Test(int a = 0)                        //定义构造函数
6      {
7          cout<<"this:"<<"<<"载入构造函数!"<<a<<endl;
8          Test::a = a;
9      }
10     Test(Test &temp)                       //定义拷贝构造函数
11     {
12         cout<<"载入拷贝构造函数!"<<endl;
13         Test::a = temp.a;
14     }
15     ~Test()                                //定义析构函数
16     {
17         cout<<"this:"<<"<<"载入析构函数!"<<"this->a<<endl;
18         cin.get();
19     }
20     operator int()                          //转换运算符
21     {
22         cout<<"this:"<<"<<"载入转换运算符函数!"<<"this->a<<endl;
23         return Test::a;
24     }
25 public:                                     //定义公有成员变量
26     int a;
27 };
28 void main()
29 {
30     Test b(99);                             //创建对象
31     cout<<"b 的内存地址"<<&b<<endl;
32     cout<<(int)b<<endl;                     //强转换
33 }
```

【运行结果】在 Visual C++ 中执行上述代码,其运行结果如图 13-9 所示。



图 13-9 类型转换运算符重载

【范例解析】上述代码中,利用转换运算符将 `Test` 类的对象强转换成了 `int` 类型并输出,



此时注意观察转换运算符函数的运行状态。发现并没有产生临时对象，证明了其与普通函数并不相同，虽然其带有 `return` 语句。



在很多情况下，类的强转换运算符还可以作为类对象加运算重载函数使用，尽管它们的意义并不相同。

13.5 小结

本章主要介绍了运算符重载的相关内容。从运算符重载的定义和特点着手，通过一个具体的范例介绍运算符重载的优点，接着简要说明了运算符重载的规则。同时，重点讲解了两种形式的运算符重载：运算符重载为成员函数和运算符重载为友元函数。针对每一种重载方式，都通过一个范例介绍了双目运算符的重载和单目运算符的重载。此外，针对几种特殊运算符，本章通过实例讲解了赋值运算符、下标运算符等重载的实现。最后，就 C++ 面向对象程序设计的基础概念和特征做了简要的概括。

13.6 习题

1. 定义计数器 `Counter` 类，对其重载运算符 `+`。

【解答】该习题主要考查双目运算符的重载实现。运算符的重载可以在类中进行定义，也可以在类外进行定义，但在类中必须进行说明。该习题首先定义一个计数器类，在公有成员中说明运算符的重载，在类外进行定义。其简要的实现代码如下所示。

```
class Counter
{
public:
    Counter();
    Counter(USHORT initialValue);
    ~Counter(){}
    USHORT GetItsVal()const { return itsVal; }
    void SetItsVal(USHORT x) {itsVal = x; }
    Counter operator+ (const Counter &);
private:
    USHORT itsVal;
};

Counter Counter::operator+ (const Counter & rhs)
{
    return Counter(it'sVal + rhs.GetItsVal());
}
```

2. 对 `Point` 类重载 `++`（自增）、`--`（自减）运算符。

【解答】该习题主要考查单目运算符 `++`（自增）、`--`（自减）的重载。读者需要注意，`++`（自增）、`--`（自减）均具有前置和后置的区别，其属于特殊运算符的重载。因此，在进行这两种运算符的重载时，需要考虑前置和后置。该习题首先定义类 `Point`，在类中对运算符重载函数进行说明，在类外对重载函数进行具体定义。其简要的实现代码如下所示。

```
class Point
{
public:
    Point& operator++();
    Point operator++(int);
    Point& operator--();
    Point operator--(int);
private:

```

```

        int _x, _y;
    };
    Point& Point::operator++()
    {
        _x++;
        _y++;
        return *this;
    }

    Point Point::operator++(int)
    {
        Point temp = *this;
        ++*this;
        return temp;
    }

    Point& Point::operator--()
    {
        _x--;
        _y--;
        return *this;
    }

    Point Point::operator--(int)
    {
        Point temp = *this;
        --*this;
        return temp;
    }

```

3. 定义 Point 类, 有成员变量 X、Y, 为其定义友元函数实现重载+。

【解答】该习题主要考查将运算符重载为友元函数的定义和调用。同样, 运算符的重载可以在类中进行定义, 也可以在类外进行定义, 但在类中必须进行说明。将运算符+进行友元函数重载, 需要定义两个操作数。其简要的实现代码如下所示。

```

class Point
{
public:
    friend Point operator+( Point& pt, int nOffset );
    friend Point operator+( int nOffset, Point& pt );
private:
    unsigned X;
    unsigned Y;
};

Point operator+( Point& pt, int nOffset )
{
    Point ptTemp = pt;
    ptTemp.X += nOffset;
    ptTemp.Y += nOffset;
    return ptTemp;
}

Point operator+( int nOffset, Point& pt )
{
    Point ptTemp = pt;
    ptTemp.X += nOffset;
    ptTemp.Y += nOffset;
    return ptTemp;
}

```



4. 设计一个程序，重载四则运算符，实现复数的算术运算。实现结果如图 13-10 所示。

【解答】该习题主要考查算术运算符的重载。复数由实部和虚部构造，因此可以定义一个复数类，然后再在类中重载复数四则运算的运算符。具体地说，可以定义类 `complex`，其中将四则运算符“+”、“-”、“*”和“/”都重载为类的成员函数，重载后的四则运算符能够实现复数的四则运算。同时声明一个友元函数用于输出运算结果。在主函数 `main()` 中创建三个对象，其中两个对象进行四则运算。其简要的实现代码如下所示。

图 13-10 运算符重载应用

```
class complex                                //定义复数类
{
public:                                       //定义公有成员
    complex()                               //定义构造函数
    {
        real=imag=0;                       //成员初始化
    }
    complex(double r, double i)             //重载构造函数
    {
        real = r;                           //成员初始化
        imag = i;
    }
    complex operator +(const complex &c);    //声明重载运算符+为成员函数
    complex operator -(const complex &c);    //声明重载运算符-为成员函数
    complex operator *(const complex &c);    //声明重载运算符*为成员函数
    complex operator /(const complex &c);    //声明重载运算符/为成员函数
    friend void print(const complex &c);     //声明函数 print 为友元函数
private:                                    //定义私有成员
    double real, imag;                      //定义双精度变量
};

inline complex complex::operator +(const complex &c) //定义重载+运算符
{
    return complex(real + c.real, imag + c.imag); //返回两个复数的实部和虚部和
}
inline complex complex::operator -(const complex &c) //定义重载-运算符
{
    return complex(real - c.real, imag - c.imag); //返回两个复数的实部和虚部差
}
inline complex complex::operator *(const complex &c) //定义重载*运算符
{
    return complex(real * c.real - imag * c.imag, real * c.imag + imag * c.real);
    //返回两个复数的实部和虚部积
}
inline complex complex::operator /(const complex &c) //定义重载/运算符
{
    return complex((real * c.real + imag * c.imag) / (c.real * c.real + c.imag * c.imag),
        (imag * c.real - real * c.imag) / (c.real * c.real + c.imag * c.imag));
    //返回两个复数的实部和虚部商
}
```



```
}  
void main()  
{  
    complex c1(2.0, 3.0), c2(4.0, -2.0), c3;    //创建 3 个复数类的对象  
    c3 = c1 + c2;                                //调用运算符+  
}
```


第 14 章 输入/输出流

经过前面章节的学习，读者已经知道，应用程序输入/输出是使用非常频繁的。一般而言，输入是为了实现程序与用户的交互，而输出是为了返回结果或给出提示信息。事实上，C++并不具有内部输入/输出的能力，即其本身并没有输入/输出语句，这样做的目的是为了最大限度地保证语言与平台的无关性。计算机语言的输入/输出功能都是与操作系统相关的，如果 C++ 为某种操作系统实现内部输入/输出功能，那它也就被限制在这个操作系统上了，这是不被希望的。本章将介绍 C++中是如何实现程序的输入和输出的。

以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为一个学习的参照。

- 了解 C++中引入标准输入/输出流的原因。
- 掌握常用标准输入/输出流对象。
- 掌握输入/输出流成员函数的使用和格式控制。

14.1 输入/输出流的引入

如果一个应用程序没有输入和输出，那它也就没有应用价值。在 C++中，输入/输出功能是通过调用该操作系统的 I/O 库来实现的。

14.1.1 printf 与 scanf 的缺陷

有过 C 语言学习经历的读者应该知道，C 语言中的输入/输出大都是由函数 scanf 和 printf 来实现的。scanf、printf 都是 C 语言标准输入/输出库函数。C 语言的标准输入/输出库函数是安全的、高效的。

函数 scanf 和 printf 都是 C 语言的标准输入/输出库中的函数(即 I/O 库函数)，C 语言的 I/O 库函数主要用来处理基本数据类型(字符、整型、浮点数等)，它们使用参数表进行数据传输，使用格式字符串指定数据类型和输入/输出格式。它们在运行时对格式字符串进行语法分析，并据此对变量进行解释。下面的范例使用了函数 scanf 和 printf 进行输入/输出。

【范例 14-1】使用了函数 scanf 和 printf 实现输入/输出。该范例实现从键盘上接收两个整型数据类型的变量值，将其按照指定的格式输出，并计算它们的输入和输出，实现代码如代码清单 14-1 所示。

代码清单 14-1

1	#include <stdio.h>	//包含头文件
2	void main()	
3	{	
4	int a;	//定义变量
5	int b;	
6	printf("Please input a and b:\n");	//输出函数
7	scanf("%d%d",&a,&b);	//输入函数
8	printf("a=%d\tb=%d\ta+b=%d\n",a,b,a+b);	//输出
9	}	

【运行结果】在 Visual C++中新建一个【C++ Source File】文件，输入如上的代码，编译无

误后运行，如果输入变量 a、b 的初值分别是 3、4，那么程序的输出结果如图 14-1 所示。

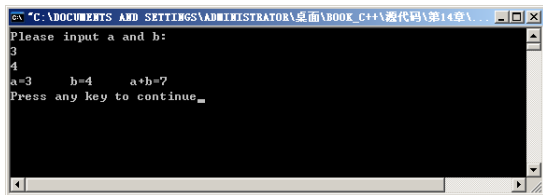


图 14-1 函数 scanf 和 printf 实现输入/输出

【范例解析】上述代码中，使用 C 语言的标准 I/O 库函数 `scanf` 和 `printf` 来实现程序的输入/输出，上述第 8 行代码指定输出变量 a、b 和 a+b 的结果，我们知道其输出值都必须是整型数据类型，其输出格式必须按照 “ ” 符号中间的格式。



注意 在 Visual C++ 编译环境下可以用 C 语言的标准 I/O 库函数 `scanf` 和 `printf`，但必须添加头文件 `stdio.h`，否则编译将无法通过。

既然使用 C 语言 I/O 库函数也能够很好地完成程序的输入/输出，那为什么还有引入 C++ 的输入/输出流呢？这是因为 C 语言 I/O 库存在以下的缺陷：

- 即使只使用了解释程序的一个功能，也要全部装载。如范例 14-1，必须要装载整个包，包括解释浮点数和字符串那部分程序段，无法减少程序的长度。
- 虽然 `printf` 族函数已经优化得很好，但是，它是在运行期间进行解释的。如果能在编译期间分析格式字符串里的变量，并根据不同的类型调用各自的函数处理，运行会快得多，而且 C++ 编译期间的类型检查会有助于用户发现错误。
- 对于 C++ 来说，输出函数 `printf` 不能被扩展是最大的缺点。用户不能通过重载函数对它进行扩展，因为重载函数要有不同类型的参数，而 `printf` 库函数把类型信息隐藏在可变参数表和格式字符串中。

14.1.2 输入/输出流简介

C++ 完全支持 C 的输入/输出系统，但由于 C 的输入/输出系统不支持类和对象，所以 C++ 又提供了自己的输入/输出系统，并通过重载运算符 “<<” 和 “>>” 来支持类和对象的输入/输出。C++ 的输入/输出系统是以字节流的形式实现的。

C++ 中的流是指数据从一个对象传递到另一个对象的操作。从流中读取数据称为提取操作，向流内添加数据称为插入操作。流在使用前要建立，使用后要删除。如果数据的传递是在设备之间进行，这种流就称为 I/O 流。C++ 专门内置了一些供用户使用的类，在这些类中封装了可以实现输入/输出操作的函数，这些类统称为 I/O 流类。

此外，流具有方向性：与输入设备相联系的流称为输入流，与输出设备相联系的流称为输出流；与输入/输出设备相联系的流称为输入/输出流，如图 14-2 所示。

前面内容提到了，C++ 没有使用 C 的输入/输出库，而是使用 `iostream` 流库。`iostream` 是通过类的继承，以及类成员函数的重载来实现的。利用类的可继承性和多态性，使 `iostream` 类库使用统一的函数接口操作标准 I/O、文件、存储块等输入/输出设备。



提示 通过函数重载，为每种内部数据类型定义了流输入/输出函数，使得用户可以用相同的格式对各种数据类型进行操作，编译程序根据数据的类型自动选择相应的输入/输出函数，不必将所有函数一并加载。

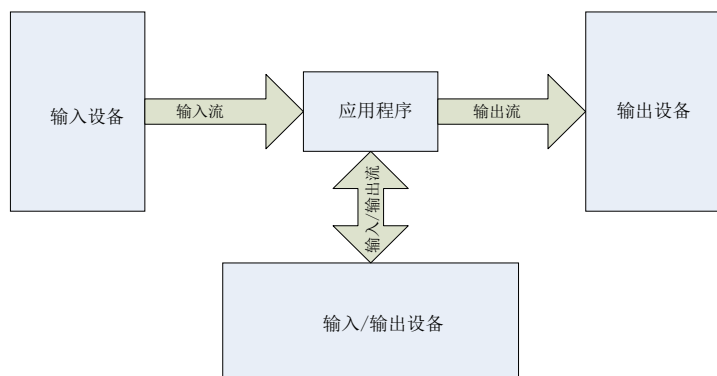


图 14-2 输入/输出流

同时，`iostream` 拥有很好的扩展性，用户通过重载还可以对自定义对象进行流的操作。因此，与标准 C 输入/输出库的各种各样的函数相比，输入/输出流更容易、更安全、更有效。

14.1.3 输入/输出流类层次

由于 C++ 的流类库是用派生方法建立起来的输入/输出类库，因此它必然有基类和派生类。C++ 中，它有两个平行的基类 `streambuf` 和 `ios`，其他的流类都是从这两个基类直接或间接派生的。使用这些流类库时，必须包含相应的头文件。

1. `streambuf` 类

`streambuf` 类是带有缓冲区的流类库，它的作用是提供物理设备的接口、缓冲或处理流的通用方式，几乎不需要任何格式。当其用做流类库中的基类时，派生以下三个流类。

- `filebuf` 类：使用文件来保存缓冲区中的字符序列。
- `strstreambuf` 类：扩展类 `streambuf` 的功能，提供在内存进行提取和插入操作的缓冲区管理。
- `onbuf` 类：扩展类 `streambuf` 的功能，用于处理输出、提供控制光标、设置颜色、定义活动窗口、清屏、清一行等功能，为输出操作提供缓冲区管理。

该类使用的缓冲区由一个字符序列和输入缓冲区指针与输出缓冲区指针组成，指针指向字符被取出或插入的位置。通常情况下，均使用这三个派生类，很少直接使用 `streambuf` 类。

2. `ios` 类

`ios` 类及其派生类为用户提供了使用流类的接口，它们均由一个指针指向 `streambuf` 类。`ios` 类及其派生类使用 `streambuf`，以及派生类来完成对错误的格式化输入/输出的检查，并且支持对 `streambuf` 类的缓冲区进行 I/O 时的格式化或非格式化转换。



提示 `ios` 作为流类库中的基类，可以派生出许多类，这些类在实际的程序中应用都较频繁，其类的层次关系如图 14-3 所示。

图 14-3 中，`ios`、`istream`、`ostream` 和上述提到的带缓冲区的流类库 `streambuf` 类构成了 C++ 中 `iostream` 输入/输出功能的基础。流是一个抽象的概念，实际进行 I/O 操作时，必须将流与一种具体的物理设备联系起来。例如，将流和键盘联系起来，当从该流中提取数据时，就是从键盘输入数据。用户也可以用 `istream`、`ostream` 等类声明自己的流对象，例如：

```
istream is; ostream os;
```

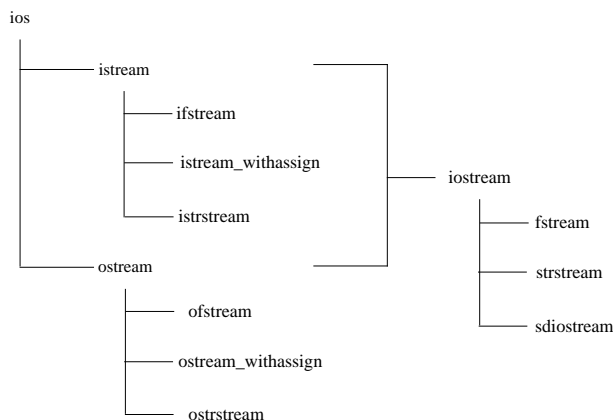


图 14-3 ios 类的层次关系



使用流类库的程序，不但可以识别标准的 I/O 设备，还可以重载运算符“<<”和“>>”，使程序识别用户自定义的类型，从而极大地提高程序的可靠性和灵活性。

14.2 标准输入/输出流

C++ 将一些常用的流类对象，如键盘输入、显示器输出、程序运行出错输出、打印机输出等，实现定义并内置在系统中，供用户直接使用。这些系统内置的用于设备间传递数据的对象称为标准流类对象，共有 4 个。

- cin 对象：与标准输入设备相关联的标准输入流。
- cout 对象：与标准输出设备相关联的标准输出流。
- cerr 对象：与标准错误输出设备相关联的非缓冲方式的标准输出流。
- clog 对象：与标准错误输出设备相关联的缓冲方式的标准输出流。

本节将一一介绍这些标准输入/输出流的使用。

14.2.1 标准输出流对象

标准输出流对象是采用 cout 对象将输出流中的数据显示在屏幕上，称为输出操作。使用 cout 对象的方法很简单，下面通过一个简单范例来理解。

【范例 14-2】标准输出流对象的使用。该范例通过引用标准输出流对象输出了一个简单的字符串，代码如代码清单 14-2 所示。

代码清单 14-2

```
1  #include <iostream>                                //注意不是 iostream.h
2  using namespace std;                                //使用命名空间
3  void main()
4  {
5      std::cout << "Hello World !" << std::endl;    //标准输出流对象
6  }
```

【运行结果】在 Visual C++ 中运行上述程序，其返回结果如图 14-4 所示。

【范例解析】上述代码中，第 5 行语句用了两次输出操作符。每个输出操作符都接受两个



操作数：左操作数必须是 ostream 对象；右操作数是要输出的值。操作符将其右操作数作为其左操作数的 ostream 对象。

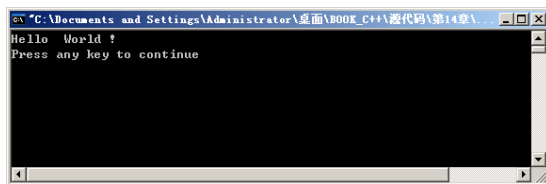


图 14-4 标准输出流对象

读者可以看到，上述代码的功能是在屏幕上输出“Hello World!”，读者应该注意到这与以前使用的程序稍有不同。事实上，在 Visual C++ 中实现同样的功能，还可以采用如下的代码实现：

```
#include <iostream.h>
void main()
{
    cout << "Hello World !" << endl;
}
```

事实上，<iostream>和<iostream.h>是不一样的，前者没有后缀。实际上，从编译器 include 文件夹里面可以看到，二者是两个文件，打开文件就会发现，里面的代码是不一样的。对于后缀为.h 的头文件，C++ 标准已经明确提出不支持了，早些的实现将标准库功能定义在全局空间里，声明在带.h 后缀的头文件里，C++ 标准为了和 C 区别开，也为了正确使用命名空间，因此规定头文件不使用后缀.h。



注意 本章示例中使用的都为<iostream.h>的形式，即没有使用命名空间，这是因为在 Visual C++ 6.0 中两种方式都支持，同时可以使得初学者和以前有 C 基础的读者更容易看懂其中的代码。

14.2.2 标准输入流对象

与标准输出流对象相对应的，标准输入流用于将键盘等输入设备中的信息送到程序中，其使用的是 cin 对象，称为输入操作。

【范例 14-3】标准输入流对象。该范例根据输出提示，接收从键盘输入的两个整型变量的值，并将输入的变量值进行简单的求和运算后将结果输出，代码如代码清单 14-3 所示。

代码清单 14-3

```
1  #include <iostream>
2  using namespace std;           //使用命名空间
3  void main()
4  {
5      std::cout << "Enter two numbers:" << std::endl;    //输出
6      int v1, v2;           //定义变量
7      std::cin >> v1 >> v2;    //标准输入流
8      std::cout << "The sum of " << v1 << " and " << v2 << " is " << v1 + v2 <<
std::endl;
9  }
```

【运行结果】在 Visual C++ 中运行上述程序，其运行结果如图 14-5 所示。

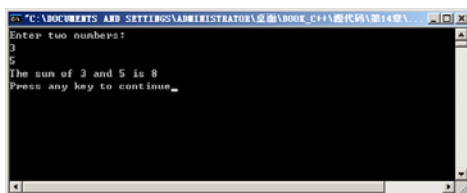


图 14-5 标准输入流对象

【范例解析】上述代码中，在输出提示语后，将读入用户输入的数据。其中，输入操作符 (>>操作符) 行为与输出操作符相似，其接受一个 `istream` 对象作为其左操作数，接受一个对象作为其右操作数，其从 `istream` 操作数读取数据并保存到右操作数中。

与输出操作符类似，输入操作符返回其左操作数作为结果。由于输入操作符返回其左操作数，可以将输入请求序列合并成单个语句，代码如下所示：

```
std::cin >> v1
std::cin >> v2
```

上述语句的输入操作效果是从标准输入中读取两个值，将第一个存放在变量 `v1` 中，第二个存放在变量 `v2` 中。



警告 上述程序中仍然使用了命名空间，读者要注意的是声明语句 “`using namespace std;`” 必须以分号 “`;`” 结束，否则编译系统将报错。

14.2.3 标准错误输出流对象

事实上，C++ 的标准错误输出流对象包括 `cerr` 对象和 `clog` 对象。其中，前者是与标准错误输出设备相关联的非缓冲方式的标准输出流，而后者是与标准错误输出设备相关联的缓冲方式的标准输出流。本节将要介绍这两种对象。

在默认方式下，标准输入设备是键盘，标准输出设备是显示器，而不论何种情况，标准输出设备总是显示器。`cin` 对象和 `cout` 对象前面已作过说明，`cerr` 对象和 `clog` 对象都是输出错误信息，它们的区别是：`cerr` 没有缓冲区，所有发送给它的出错信息都被立即输出；`clog` 对象带有缓冲区，所有发送给它的出错信息都先放入缓冲区，当缓冲区满时再进行输出，或通过刷新流的方式强迫刷新缓冲区。由于缓冲区会延迟错误信息的显示，所以建议使用 `cout` 对象。

和标准输出流 `cout` 对象一样，标准错误输出流对象 `cerr` 也是一个 `ostream` 对象。两者之间的区别在于：操作系统重定向只影响 `cout`，而不影响 `cerr`，`cerr` 对象用于错误消息。因此，如果将程序输出重定向到文件，并且发生了错误，则屏幕上仍然会出现错误消息。在 UNIX 系统中，可以分别对 `cout` 和 `cerr` 进行重定向，命令行操作符 `>` 用于对 `cout` 进行重定向，操作符 `2>` 对 `cerr` 进行重定向。

【范例 14-4】`cerr` 对象。该范例同时采用了标准输出流对象 `cout` 和标准错误输出流对象 `cerr`，读者可以查看其运行时是否有区别，代码如代码清单 14-4 所示。

代码清单 14-4

```
1  #include <iostream>
2  using namespace std;           //使用命名空间
3  void main()
4  {
5      cout << "cout" << endl;    //使用 cout 对象
6      cerr << "cerr" << endl;    //使用 cerr 对象
7  }
```



【运行结果】在 Visual C++中运行上述程序，其运行结果如图 14-6 所示。

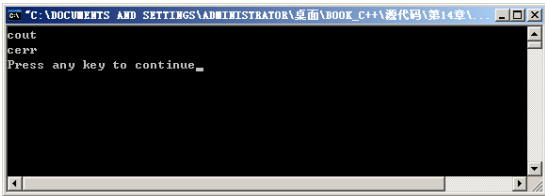


图 14-6 cerr 对象

【范例解析】读者可以看到，此时标准错误输出流对象 cerr 与标准输出流对象 cout 具有同样的功能，即输出信息到屏幕。但是，如果读者编译上述程序后，进入到 CMD 命令行中，在命令行里执行 14-4 2>14-4.log，然后打开 14-4.log 日志文件，可以看到不同的流进行不同的重定向，2>14-4.log 就是将标准错误输出重定向到 14-4.log 中，如图 14-7 所示。

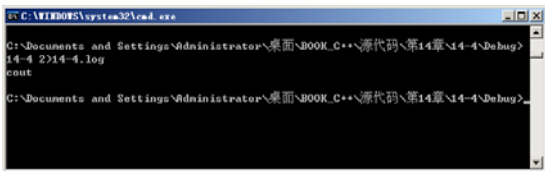


图 14-7 cerr 对象与 cin 对象的比较

此外，cout 对象也能输出错误信息，但当用户把标准输出设备定向为其他设备时，cerr 对象仍然把信息发送到显示器。这些标准流类对象都包含在头文件 iostream.h 中，使用时应包含该头文件。

与 cerr 对象不同的是，clog 是与标准错误输出设备相关联的缓冲方式的标准输出流，即其是有缓冲的，错误信息不会立即输出在屏幕中。至于 clog 对象的应用，它与 cerr 对象的应用方法基本相似，读者可以试着自己实现。

提示 fstream.h 和 strstream.h 中都包含了 iostream.h，所以如果使用标准输入/输出(控制台 I/O)，只要包含 iostream.h 头文件即可；如果使用 fstream 或者 strstream，只要包含相应的 fstream.h 和 strstream.h 即可。

14.3 输入/输出流成员函数

C++中，输入/输出流除了可以使用前面介绍的输入/输出流对象外，类 istream 还有三个从流中进行非格式化抽取的成员函数：get()、getline()和 read()。本节将介绍其中使用较多的两个成员函数。

14.3.1 get()函数：输出字符串

事实上，根据 get()函数的具体应用范围的不同，成员函数 get()的用法可以通过函数重载去实现不同功能，如表 14-1 所示。

表 14-1 get()函数的用法

形 式	说 明
int get()	从流中抽取单个字符并返回



续表

形 式	说 明
istream& get(char*,int,char)	从流中抽取字符直到终止符（默认为'\n'）或者抽取字符达到第二个参数给定的数量或者已到文件尾，将其存储在第一个参数指定的字符数组里
istream& get(char &)	从流中抽取单个字符并存入引用变量中
istream& get(streambuf &,char)	从流中取得字符存入 streambuf 对象，直到终止符或文件尾

【范例 14-5】get()函数的应用。该范例实现 get()函数的几种不同用法，其分别根据用户在键盘上的输入来输出一个字符、一个字符串等，代码如代码清单 14-5 所示。

代码清单 14-5

1	#include <iostream.h>	
2	void main()	
3	{	
4	char s1[80],s2[80];	//定义字符数组
5	cout<<"请键入一个字符:" ;	
6	cout<<cin.get()<<endl;	
7	cin.get();	//get()函数调用，接收用户输入
8	cout<<"请输入一行字符串:" ;	
9	for(int i=0 ; i<80 ;i++)	//输入字符数组元素
10	{	
11	cin.get(s1[i]) ;	
12	if(s1[i]=='\n')	//判断是否回车
13	{	
14	s1[i]='\0' ;	
15	break ;	//跳出循环
16	}	
17	}	
18	cout<<s1<<endl ;	//输出
19	cout<<"请输入一行字符串:" ;	
20	cin.get(s2,80) ;	//get()函数调用
21	cout<<s2<<endl ;	//输出
22	}	

【运行结果】在 Visual C++中运行上述程序，其运行结果如图 14-8 所示。

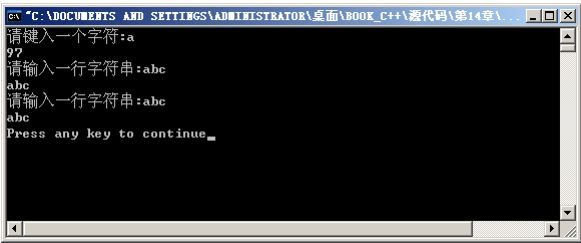


图 14-8 get()函数的应用

【范例解析】读者可以看到，上述代码通过 get()函数来实现取一个字符，通过定义字符数组和 for 循环来实现取一个字符串，最后通过重载 get()函数（即使用表 14-1 中的重载形式）实现取一个字符串并将其结果输出。

注 get()函数有许多形式，其重载后的形式可包含参数，其返回类型也可为字符串，在实际程序中一般用不带参数的形式更多。



14.3.2 getline()函数：获取字符串

getline()函数也可以实现一个字符串的取出，其定义格式为：

```
istream& getline(char*,int,char)
```

该函数从流中抽取字符直到终止符（默认为'\n'），或者抽取字符达到参数给定的数量-1；或者已到文件尾，将其存储在第一个参数指定的字符数组里。如果发现终止符，其从流中提取终止符，但只是抛弃掉，并不把它存在结果缓冲区里。

【范例 14-6】getline()函数的应用。该范例通过 getline()函数将字符串取出，读者可将其与上述的 get()函数相比较，看其应用范围是否有不同，代码如代码清单 14-6 所示。

代码清单 14-6

```
1  #include <iostream.h>
2  void main()
3  {
4      char s1[80];                //定义字符串
5      cout<<"请输入一个字符:" ;
6      cout<<cin.get()<<endl;      //输出
7      cin.get() ;                 //调用 get()函数
8      cout<<"请输入一行字符串:" ;
9      cin.getline(s1,80) ;        //调用 getline()函数
10     cout<<s1<<endl;             //输出字符串
11 }
```

【运行结果】在 Visual C++中运行上述程序，其运行结果如图 14-9 所示。

【范例解析】上述程序中，其分别使用了 get()函数实现一个字符的输出，使用 getline()函数实现一个字符串的输出。

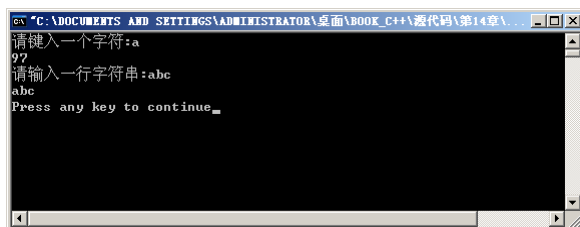


图 14-9 getline()函数的应用



提示 事实上，get 和 getline 都可以给参数赋值，在输入字符和字符串的时候，C++会有一个结束字符“\0”，用 get()函数输入的时候，会在参数中留下这个字符，而用 getline()函数则不会留下，这是两者主要的区别。

14.4 输入/输出的格式控制

读者知道，C++仍可使用 C 中的 printf()和 scanf()进行格式化控制。同时，C++又提供了两种格式化控制的方法：一种是使用 ios 类中的有关格式控制的成员函数，另一种是使用被称为格式控制符的特殊类型的函数。

14.4.1 用 ios 类的成员函数进行格式控制

一般来说，ios 类的成员函数进行格式控制主要是通过对格式状态字、域宽、填充字符和

输出精度操作来完成的。

1. 状态字

状态字也称状态标志字，其类型为 `long int`。它是在 `ios` 类的 `public` 部分定义了一个枚举，此枚举类型的每个成员分别定义状态字的一个位，每个位称为状态标志位，该枚举的定义如下：

```
enum
{
    skipws      = 0x0001    //跳过输入中的空白，用于输入
    left        = 0x0002    //左对齐输出，用于输出
    right       = 0x0004    //右对齐输出，用于输出
    internal    = 0x0008    //在符号位和其指示符后填入字符，用于输出
    dec         = 0x0010    //转换基数为十进制，用于输入或输出
    oct         = 0x0020    //转换基数为八进制，用于输入或输出
    hex         = 0x0040    //转换基数为十六进制，用于输入或输出
    showbase    = 0x0080    //输出时显示其指示符，用于输入或输出
    showpoint   = 0x0100    //输出时显示小数点，用于输出
    uppercase   = 0x0200    //十六进制输出时，表示制式的和表示数值的一律为大写，用于输出
    showpos     = 0x0400    //正整数前显示“+”符号，用于输出
    scientific  = 0x0800    //用科学表示法显示浮点数，用于输出
    fixed       = 0x1000    //用定点形式显示浮点数，用于输出
    unitbuf     = 0x2000    //在输出操作后立即刷新所有流，用于输出
    stdio       = 0x4000    //在输出操作后刷新 stdout 和 stderr，用于输出
}
```

这些枚举元素的值的共同特点是，使状态标志字二进制表示中的不同位为 1，使它们共同组成状态标志字存放在数据成员 `long x_flags` 中。这些状态之间是活的关系，可以几个并存。

2. 用 `ios` 类的成员函数进行格式控制

`ios` 类提供了几个用于控制输入/输出格式的成员函数，其中的参数 `flags` 是状态控制字，存放在 `ios` 类中的数据成员 `long x_flags` 中。主要成员函数的使用方法如下。

(1) 设置状态标志，指用函数 `setf()` 将状态字标志置“1”，函数原型为：

```
long ios::setf(long flags)
```

使用时的一般调用形式为：

```
流对象.setf(ios::状态标志);
```

例如：

```
istream isobj;
ostream osobj;
isobj.setf(ios::skipws);
osobj.setf(ios::right|ios::showpos|ios::dec);
```

(2) 清除状态标志指用函数 `unsetf()` 将某一状态标志置“0”，函数原型为：

```
long ios::unsetf(long flags)
```

使用时的一般调用形式为：

```
流对象.unsetf(ios::状态标志);
```

(3) 取状态标志指用函数 `flags()` 返回状态标志字，有带参数和不带参数两种形式，它们的函数原型为：

```
long ios::flags()
long ios::flags(long flags)
```

使用时的一般调用形式为：



```
流对象.flags();
流对象.flags(ios::状态标志);
```



注意 不带参数时返回当前的状态标志字；带参数时将状态字设置为 flags，并返回设置前的状态标志字。flgs()与 setf()的区别是：setf()是在原有的基础上追加设置，不改变原有设置；flgs()使用新的设置覆盖原有的设置，改变了原有设置。

【范例 14-7】使用成员函数控制输入/输出格式。该范例输出了许多字符和字符串，其中使用到了多种 cout 对象的成员函数来控制格式，代码如代码清单 14-7 所示。

代码清单 14-7

```
1  #include<iostream.h>
2  void main()
3  {
4      cout<<"x_width="<<cout.width()<<endl;           //设置默认域宽
5      cout<<"x_fill="<<cout.fill()<<endl;             //填充默认字符空格
6      cout<<"x_precision="<<cout.precision()<<endl;   //设置默认精度
7      cout<<123<<"    "<<123.456789<<endl;           //输出
8      cout<<"-----"<<endl;
9      cout.width(10);                                   //设置域宽 10
10     cout.precision(3);                                //设置精度 3
11     cout<<123<<"    "<<123.456789<<endl;
12     cout<<"x_width="<<cout.width()<<endl;           //调用 width()函数
13     cout<<"x_fill="<<cout.fill()<<endl;             //调用 fill()函数
14     cout<<"x_precision="<<cout.precision()<<endl;   //调用 precision()函数
15     cout<<"-----"<<endl;
16     cout.width(10);
17     cout.fill('*');                                   //设置填充 "*"
18     cout<<123<<"    "<<123.456789<<endl;
19     cout.width(10);
20     cout.setf(ios::left);                             //设置状态标志
21     cout<<123<<"    "<<123.456789<<endl;
22     cout<<"x_width="<<cout.width()<<endl;
23     cout<<"x_fill="<<cout.fill()<<endl;
24     cout<<"x_precision="<<cout.precision()<<endl;
25 }
```

【运行结果】在 Visual C++中运行上述程序，其运行结果如图 14-10 所示。

```

C:\DOCUMENTS AND SETTINGS\ADMINISTRATOR\桌面\BOOK_C++\源代码\第14章\...
x_width=0
x_fill=
x_precision=6
123    123.457
-----
      123    123
x_width=0
x_fill=
x_precision=3
*****123    123
123***** 123
x_width=0
x_fill=*
x_precision=3
Press any key to continue_

```

图 14-10 使用成员函数控制输入/输出格式

【范例解析】读者可以看出，使用成员函数可以按照用户自己的需求对输出的格式进行控制，其主要包括对输出域宽、精度和填充字符等方面的设置。上述代码中，首先采用系统的默认设置来进行控制，接着对这几方面进行修改，查看其输出结果。



14.4.2 使用格式控制符进行格式控制

14.4.1 节使用成员函数控制输入/输出格式时, 每个函数的调用都要写一条语句, 它们还不能直接嵌入到输入/输出语句中, 这使得使用很不方便。为此, C++ 提供了另外一种输入/输出格式的控制方法, 即使用称为格式控制符的特殊函数。事实上, 这部分内容在第 4 章中已经提到了, 此处将其放在输出流中系统地讲解相关的知识, 读者可与前面的内容相对比。

1. 预定义的格式控制符

预定义的格式控制符可以直接嵌入到输入/输出语句中, 完成类似于 `ios` 类中控制输入/输出格式的成员函数的功能。一般来说, C++ 的预定义格式控制符如表 14-2 所示。

表 14-2 预定义的格式控制符

格式控制符	功 能
<code>dec</code>	以十进制形式输入/输出整数, 用于输入或输出
<code>hex</code>	以十六进制形式输入/输出整数, 用于输入或输出
<code>oct</code>	以八进制形式输入/输出整数, 用于输入或输出
<code>ws</code>	用于输入时跳过开头的空白符, 仅用于输入
<code>endl</code>	插入一个换行符并刷新输出流, 仅用于输出
<code>ends</code>	插入一个空字符, 用来结束一个字符串, 仅用于输出
<code>flush</code>	刷新一个输出流, 仅用于输出
<code>setbase(int n)</code>	把转换基数设置为 <code>n</code> (<code>n=0,8,10,16</code>), 默认值为 0 (十进制)
<code>resetiosflags(long f)</code>	关闭由参数 <code>f</code> 指定的格式标志, 用于输入或输出
<code>setiosflags(long f)</code>	设置由参数 <code>f</code> 指定的格式标志, 用于输入或输出
<code>setfill(int c)</code>	设置 <code>c</code> 为填充字符, 默认为空格, 用于输入或输出
<code>setprecision(int n)</code>	设置小数位数, 默认为 6 位, 用于输入或输出
<code>setw(int n)</code>	设置域宽, 用于输入或输出

其中, 格式控制符 `setiosflags(long f)` 和 `resetiosflags(long f)` 中的格式标志与 `ios` 类中控制输入/输出格式的成员函数所用的标志基本相同, 此处不再说明。

2. 预定义格式控制符的使用

预定义格式控制符分为带参数和不带参数的两种, 带参数的在头文件 `iomanip.h` 中定义, 不带参数的在头文件 `iostream.h` 中定义。使用它们时, 程序中应包含相应的头文件。格式控制符被嵌入到输入/输出语句中控制输入/输出的格式, 而不是执行输入/输出操作。

【范例 14-8】 使用预定义的格式控制符控制输入/输出格式。该范例使用预定义的控制符进行输入/输出格式控制, 代码如代码清单 14-8 所示。

代码清单 14-8

```
1  #include<iostream.h>
2  #include<iomanip.h>           //包含格式控制头文件
3  void main()
4  {
5      cout<<setw(10)<<987<<654<<endl;           //设置域宽为 10
6      cout<<987<<setiosflags(ios::scientific)<<setw(15)<<987.654321<<endl;
7      cout<<987<<setw(10)<<hex<<987<<endl;       //设置域宽和十六进制输出
8      cout<<987<<setw(10)<<oct<<987<<endl;        //设置域宽和八进制输出
9      cout<<987<<setw(10)<<dec<<987<<endl;        //设置域宽和十进制输出
```



```

10
cout<<resetiosflags(ios::scientific)<<setprecision(3)<<987.654321<<endl;
11     cout<<setiosflags(ios::left)<<setfill(' '<<setw(7)<<987<<endl;
12     cout<<setiosflags(ios::right)<<setfill('#')<<setw(7)<<987<<endl;
13 }

```

【运行结果】在 Visual C++ 中运行上述程序，其运行结果如图 14-11 所示。

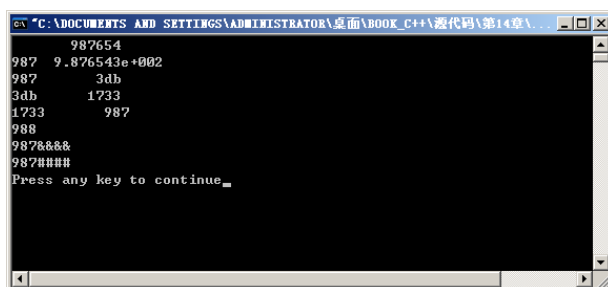


图 14-11 预定义格式控制符应用

【范例解析】上述代码中，其使用了 `setw` 控制符设置域宽、`setiosflags` 控制符设置标识位等，分别输出了一个数的十进制、十六进制和八进制的值，此外对一些输出使用 `setfill` 控制符进行了字符填充的操作。



注意 格式控制符 `setw` 只对最靠近它的输出起作用，格式控制符 `dec`、`oct` 和 `hex` 的作用则一直保持到重新设置为止，格式控制符 `setprecision` 在输出时作四舍五入处理。

3. 自定义的格式控制符

除了上述系统定义的格式控制符外，用户还可自定义格式控制符。一般来说，为输出流自定义格式控制符的一般形式为：

```

ostream &格式控制符名(ostream &stream)
{
    //自定义代码
    return stream;
}

```

而为输入流自定义格式控制符的一般形式为：

```

istream &格式控制符名(istream &stream)
{
    //自定义代码
    return stream;
}

```

上述定义中，格式控制符名和格式控制符代码由用户给出，除 `stream` 可使用其他标识符外，其余的都照原样写上。特别注意不要丢了返回语句，它是自定义格式控制符的关键。

```
return stream;
```

【范例 14-9】使用自定义格式控制符控制输入/输出格式。该范例使用自定义的控制符进行输入/输出格式的控制，代码如代码清单 14-9 所示。

代码清单 14-9

```

1  #include<iostream.h>
2  #include<iomanip.h>                                //包含头文件
3  ostream &output1(ostream &out)                    //定义格式

```

```

4  {
5      cout.setf(ios::left);                //从左输出
6      cout<<setw(10)<<dec<<setfill('*');    //域宽为 10, 十进制输出, 填充*
7      return out;
8  }
9  void main()
10 {
11     cout<<345<<endl;                    //输出默认格式的字符串
12     cout<<output1<<345<<endl;           //自定义格式控制符输出
13 }

```

【运行结果】在 Visual C++ 中运行上述程序，其运行结果如图 14-12 所示。

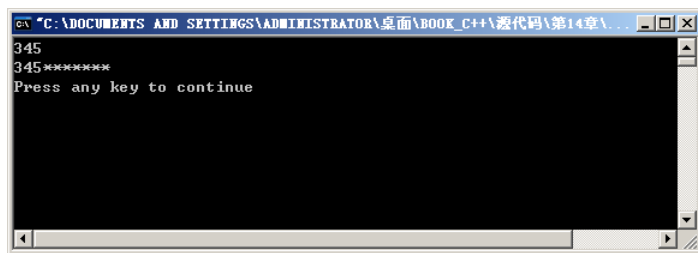


图 14-12 自定义格式控制符的应用

【范例解析】上述代码中，定义了一个函数 `output1` 用于格式控制，该函数从左侧输出域宽为 10 的十进制数值，若位数不够则在其后用 “*” 填充。在 `main()` 函数中首先输出没有格式控制的数值 345，接着输出采用了自定义格式控制符 `output1` 的同一数值 345，读者可对比查看其区别。



注意 自定义一个函数为格式控制符，其返回值必须是一个 `ostream` 类类型，否则调用时编译系统将出现错误信息。

14.5 用户自定义数据类型的输入/输出

用户自定义数据类型的输入/输出，是通过重载运算符 “<<” 和 “>>” 实现的，本节将分别介绍这两种运算符的重载。

14.5.1 重载输出运算符 “<<”

重载输出运算符 “<<” 也称为插入运算符，用做用户自定义类型的输出。定义运算符 “<<” 重载函数的一般形式为：

```

ostream &operator<<(ostream &stream, 类名 对象名)
{
    //操作代码
    return stream;
}

```

其中，第一个参数 `stream` 是对 `ostream` 对象的引用，必须是输出流，它可以是其他合法的标识符，但必须与 `return` 后面的标识符相同；第二个参数接收将被输出的对象。

需要注意的是，在重载输出运算符 “<<” 时，要注意如下的问题：

- 重载输出运算符函数不能是类的成员函数，必须是类的非成员函数。
- 作为非成员函数，重载输出运算符函数不能访问类的私有成员，为解决这个问题，应把重载输出运算符函数定义为类的友元函数。



【范例 14-10】重载输出运算符“<<”的实现。该范例将输出运算符“<<”重载为友元函数，重载后的运算符实现输出类类型中成员变量 x 和 y 的值，实现代码如代码清单 14-10 所示。

代码清单 14-10

```

1  #include<iostream.h>
2  class point                                //定义类
3  {
4      int x,y;                                //定义私有成员
5  public:
6      point()                                //定义构造函数
7      {
8          x=0;
9          y=0;
10     }
11     point(int xx,int yy)                    //重载构造函数
12     {
13         x=xx;
14         y=yy;
15     }
16     friend ostream &operator<<(ostream &stream, point obj); //重载运算符为友元函数
17 };
18 ostream &operator<<(ostream &stream, point obj) //定义该友元函数
19 {
20     stream<<"x="<<obj.x<<"    "<<"y="<<obj.y<<endl;
21     return stream;
22 }
23 void main()
24 {
25     point p1(1,2),p2(3,4);                //创建对象
26     cout<<p1<<p2;                          //调用输出运算符
27 }
```

【运行结果】在 Visual C++中运行上述程序，其运行结果如图 14-13 所示。

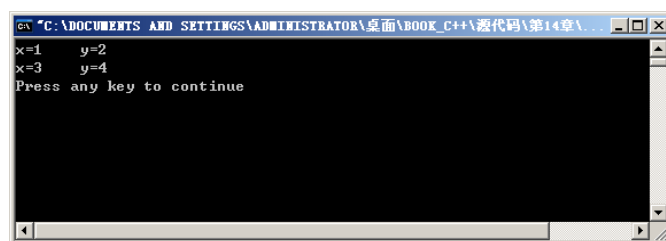


图 14-13 重载输出运算符

【范例解析】上述代码中，对输出运算符“<<”实现了重载，重载后的输出运算符可以输出类成员变量 x 和 y 的值。在主函数 main()中创建了两个对象 p1 和 p2，并对其进行了初始化。在调用 cout 对象进行输出时，由于“<<”运算符被重载，其可以输出类的两个成员变量的值。

14.5.2 重载输入运算符“>>”

重载输入运算符“>>”也称为提取运算符，用户用于自定义类型的输入。定义运算符“>>”重载函数的一般形式为：

```

istream &operator>>(istream &stream,类名 对象名)
{
```

```

//操作代码
return stream;
}

```

其中, 第一个参数 `stream` 是对 `istream` 对象的引用, 必须是输入流, 它可以是其他合法的标识符, 但必须与 `return` 后面的标识符相同。第二个参数是一个引用, 前面的 “&” 不能省略。



注意 在使用重载输入运算符 “>>” 中需要注意如下问题, 重载输入运算符函数不能是类的成员函数, 必须是类的非成员函数; 作为非成员函数, 重载输入运算符函数不能访问类的私有成员, 为解决这个问题, 应把重载输入运算符函数定义为类的友元函数。

【范例 14-11】 重载输入运算符 “>>”。该范例中重载了输入运算符 “>>” 和输出运算符 “<<”, 其中重载后的输入运算符 “>>” 实现输入类的两个成员变量, 重载后的输出运算符 “<<” 实现输出类的成员变量值, 它的实现代码如代码清单 14-11 所示。

代码清单 14-11

```

1  #include<iostream.h>
2  class point                                //定义类
3  {
4      int x,y;                                //定义私有成员
5  public:
6      point()                                //定义构造函数
7      {
8          x=0;
9          y=0;
10     }
11     point(int xx,int yy)                    //重载构造函数
12     {
13         x=xx;
14         y=yy;
15     }
16     friend ostream &operator<<(ostream &output, point obj);
                                           //声明重载运算符为友元函数
17     friend istream &operator>>(istream &input, point &obj);
18 };
19 ostream &operator<<(ostream &output, point obj)    //定义友元函数
20 {
21     output<<"x="<<obj.x<<"    "<<"y="<<obj.y<<endl; //输出
22     return output;
23 }
24 istream &operator>>(istream &input, point &obj)    //定义友元函数
25 {
26     cout<<"请输入x,y的值: "<<endl;
27     input>>obj.x;                                //输入
28     input>>obj.y;
29     return input;
30 }
31 void main()                                    //主函数
32 {
33     point p(10,20);                                //创建对象
34     cout<<p;                                        //调用输出运算符
35     cin>>p;                                        //调用输入运算符
36     cout<<p;
37 }

```

【运行结果】 在 Visual C++ 中运行上述程序, 其运行结果如图 14-14 所示。

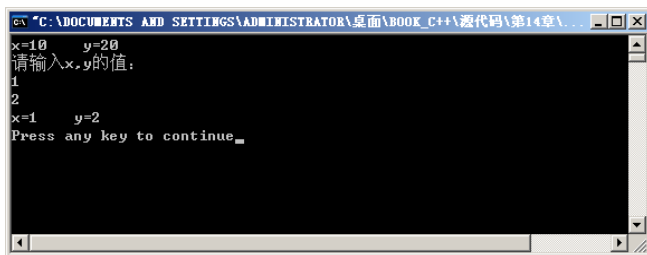


图 14-14 重载输入运算符

【范例解析】上述代码中，重载后的输出运算符“<<”和输入运算符“>>”都实现了对类的输出/输入。在主函数 main() 中创建一个对象 p 后，调用 cout 对象进行输出/输入，其使用重载后的输出/输入运算符，实现了类的输出/输入。



注意 在重载输入运算符时，其对象参数必须是一个引用，即其前面必须加“&”符号，否则输入的值不会传递到类的成员变量中，读者可自行检验。

14.6 小结

本章主要介绍了 C++ 的输入/输出流的相关内容。首先由 C 语言中的输入/输出函数 scanf() 和 printf() 的缺陷引出 C++ 中的输入/输出流，接着详细讲解了 C++ 的标准输入/输出流，并就 C++ 的输出格式控制符做了详细介绍。读者学习完本章后，必须能在实际程序中使用不同的输入/输出流进行 I/O 操作。此外，本章就输入/输出运算符“>>”和“<<”进行了重载，使其能够进行类之间的输入/输出，这也是本章的重点和难点之一。

14.7 习题

1. 分析下列程序的输出结果：

```
#include<iostream.h>
int main()
{
    char buf[]=" 12345"
    int I,j;
    istream s1(buf);
    s1>>I;
    istream s2(buf,3);
    s2>>j;
    cout<<I+j<<endl;
}
```

【解答】该习题主要考查输入流类的问题。上述程序中定义了字符数组并为其赋初值 12345，同时创建了输入流对象 s1，通过该对象来对字符数组进行操作，最后将变量 I 和 j 中的值进行相加。根据前面学习的内容，读者可以分析出，该程序的输出结果为 12468。

2. 编写一个程序，统计从键盘上输入每一行字符的个数，从中选出最长的行的字符个数，统计共输入多少行。

【解答】该习题主要考查输入流对象的成员函数的使用。可以在接收用户输入的同时就通过对象 cin 的成员函数 gcount 进行字符个数的统计，同时通过一个循环语句接收用户的输入，使用 cin 的成员函数 getline() 判断用户的输入是否完成。其简要的实现代码如下所示。

```
while (cin.getline(buf,SIZE))
{
```

```

int count=cin.gcount();
lcnt++;
if(count>lamx) lamx=xount;
cout<<"Line#"<<lcnt<<"\t"<<"chars read:"<<count<<endl;
cout.write(buf,count).put('\n').put('\n');
}

```

3. 有一元二次方程 $ax^2+bx+c=0$, 其一般解为 $x_1, x_2 = \dots$ 但若 $a=0$, 或 $b^2-4ac<0$ 时, 用此公式出错。编写一个 C++ 程序, 从键盘输入 a, b, c 的值, 求 x_1 和 x_2 。如果 $a=0$ 或 $b^2-4ac<0$, 输出出错信息。

【解答】该习题主要考查 `cerr` 和 `clog` 等标准输出流对象的使用。该习题要求在 $a=0$ 或 $b^2-4ac<0$ 时不能使用求根公式, 并输出错误信息, 此处即可使用 `cerr` 输出流来实现。判断是否出错可通过分支结构的 `if` 语句来实现。其简要的实现代码如下所示。

```

if (a==0)
    cerr<<"a is equal to zero,error!"<<endl; //将出错信息插入 cerr, 屏幕输出
else
    if ((disc=b*b-4*a*c)<0)
        cerr<<"disc=b*b-4*a*c<0"<<endl; //将出错信息插入 cerr 流, 屏幕输出
    else
        {cout<<"x1="<<(-b+sqrt(disc))/(2*a)<<endl;
         cout<<"x2="<<(-b-sqrt(disc))/(2*a)<<endl; }

```

4. 分析下列程序的输出结果。

```

#include<iostream>
#include<fstream.h>
#include<strstream.h>
const int N=80;
int main()
{
    char buf[N];
    ostrstream out1(buf,sizeof(buf));
    int a=50;
    for(int i=0;i<6;i++,a+=10)
        out1<<"a="<<a<<" ";
    out1<<"\n 0'";
    cout<<"Buf:"<<buf<<endl;
    double PI=3.1415926;
    out1.setf(ios::fixed|ios::showpoint);
    out1.seekp(0);
    out1<<"the value of pi="<<PI<<"\n 0'";
    cout<<buf<<endl;
    char *pstr=out1.str();
    cout<<pstr<<endl;
}

```

【解答】该习题主要考查输出流的实现。上述程序段定义了输出流对象 `out1`, 并通过循环语句将变量 `a` 中的值输出, 同时调用输出流的格式控制函数 `setf` 和 `seekp` 函数对输出格式进行控制, 并输出一个浮点型变量的值。根据循环语句和输出格式, 其输出结果如下:

```

Buf:a=50;a=60;a=70;a=80;a=90;a=100;
The value of pi is 3.14159265
The value of pi is 3.14159265

```

5. 设计一个程序, 根据用户输入的学生类基本信息, 将输入存储到学生类中后, 输出该学生类的所有基本信息, 如图 14-15 所示。



图 14-15 输入/输出流的应用

【解答】该程序段需要通过输入/输出运算符“>>”和“<<”直接实现对类的对象进行输入/输出操作，而不是对于单一变量的操作。这就需要对运算符“>>”和“<<”进行了重载，重载的定义实现对类的成员进行输入/输出。在主函数 main() 中，创建对象 one 后，直接对该对象进行输入/输出操作，自动调用重载后的输入/输出运算符。其简要的实现代码如下所示。

```
class Cstudent //定义学生类
{
public: //定义公有成员
    friend ostream& operator<<(ostream& os,Cstudent stu); //重载输入/输出符
    friend istream& operator>>(istream& is,Cstudent& stu); //定义私有成员
private: //定义私有成员
    char strName[10]; //姓名
    char strID[10]; //学号
    int fScore[3]; //三门成绩
};

ostream& operator<<(ostream &os,Cstudent stu) //重载输出运算符
{
    os<<endl<<"输入的学生信息如下:"<<endl<<"姓名:"<<stu.strName<<endl<<"学号:"<<stu.strID<<endl;
    os<<"三门成绩分别为:"<<stu.fScore[0]<<"\t" <<stu.fScore[1]<<"\t" <<stu.fScore[2]<<endl; //输出
    return os; //返回对象
}

istream& operator>>(istream& is,Cstudent& stu) //重载输入运算符
{
    cout<<"请输入学生信息:"<<endl<<"姓名:"; //输入提示
    is>>stu.strName; //输入学生姓名
    cout<<"学号:"; //输入提示
    is>>stu.strID; //输入学生学号
    cout<<"三门成绩:"; //输入提示
    is>>stu.fScore[0]>>stu.fScore[1]>>stu.fScore[2]; //输出成绩
    return is; //返回对象
}

void main() //主函数
{
    Cstudent one; //创建对象
    cin>>one; //调用重载后的输入运算符
    cout<<one; //调用重载后的输出运算符
}
```

第四篇 C++高级特性篇

第 15 章 文件

在实际的程序设计中，经常会遇到大批量数据的输入/输出，如果只通过用户操作界面单一地交互，执行效率无疑是很低的。为解决这个问题，人们引入了文件的概念。文件在数据的输入和输出中往往充当缓冲区的作用，它用于暂存输入/输出的数据。本章将详细介绍文件的相关操作和不同种类文件的基本操作。

以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为学习的参照。

- 了解文件和流的概念。
- 掌握文件的打开与关闭操作。
- 掌握顺序文件和随机文件的读写及其应用。

15.1 文件和流

文件是一系列字符数据的有序集合，按组织形式可分为文本文件和二进制文件两种。C++的文件把数据看作一连串的字符，而不考虑记录的界限，认为它是一个字符流或二进制流，称为流式文件，增加了处理的灵活性。

15.1.1 文件概述

文件是信息的集合，通常是指记录在外部存储介质（如磁盘等）上的信息集合。例如，用 Word 或 Excel 编辑制作的文档或表格就是一个文件，将其存放在磁盘上就是一个磁盘文件，输出到打印机上就是一个打印机文件。文件通常存放在磁盘上，通过“路径”指明其在磁盘上的位置。“路径”是由目录和文件名组成的。在程序设计中，使用到文件的概念是出于以下三个方面的考虑：

- 文件是使一个程序可以对不同的输入数据进行加工处理、产生相应输出结果的常用手段。
- 使用文件可以方便用户，提高上机效率。
- 使用文件可以不受内存大小的限制。

在计算机中，文件随着分类标准的不同可分为不同的类型。按照文件的存取方式及其组成结构来分可以分为两种类型。

- 顺序文件：结构较简单，文件中的记录一个接一个地存放。在这种文件中，只知道第一个记录的存放位置，其他记录的位置无从知道。当要查找某个数据时，只能从文件头开始，一个记录一个记录地顺序读取，直到找到为止。
- 随机文件：又称直接存取文件，简称随机文件或直接文件。随机文件的每个记录都有一个记录号，即在写入数据时只要指定记录号，就可以把数据直接存入指定位置。而在读取数据时，只要给出记录号，就可直接读取。

按照文件的数据编码方式来分可以分为以下两种。

- 文本文件：又称 ASCII 码文件，其以 ASCII 方式保存文件，可用字处理软件建立和修



改（必须以纯文本文件保存）。

- 二进制文件：不能用普通的字处理软件编辑，占空间较小。

前面提到了，文件在程序中为输入设备和输出设备承担着缓冲的功能，一般程序中，文件在程序中的作用如图 15-1 所示。

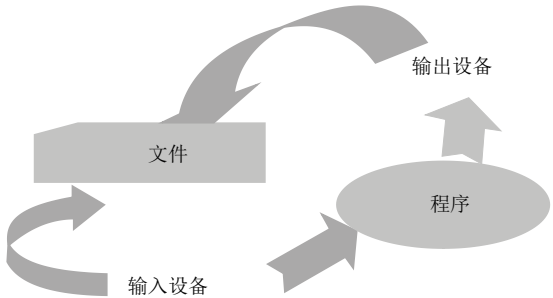


图 15-1 文件功能

提示 顺序文件组织比较简单，占空间少，容易使用，但维护困难，适用于有一定规律且不经常修改的数据。而随机文件的优点是数据存取较为灵活、方便，速度快，容易修改，主要缺点是占空间较大，数据组织复杂。

15.1.2 文件流类

前面提到了，在 C++ 中有一个 `stream` 类，所有的输入/输出（I/O）都是以这个“流”类为基础的，类 `stream` 有两个重要的运算符。

- 流输出运算符（<<）：向流输出数据。比如说，系统有一个默认的标准输出流（`cout`），一般情况下就是指的显示器，所以，语句 `cout<<"Write Stdout"<<endl;`就表示把字符串“Write Stdout”和换行控制符输出到标准输出流。
- 流输入运算符（>>）：从流中输入数据。比如说，系统有一个默认的标准输入流（`cin`），一般情况下就是指的键盘，所以，语句 `cin>>x;`就表示从标准输入流中读取一个指定类型（即变量 `x` 的类型）的数据。

文件流是 I/O 中非常重要的一个内容，它的输入是指从磁盘文件流向内存，它的输出是指从内存流向磁盘。C++ 中提供了三个文件流类：`ofstream`、`ifstream`、`fstream`，其中，`fstream` 是 `ofstream` 和 `ifstream` 多重继承的子类，文件流类的关系如图 15-2 所示。

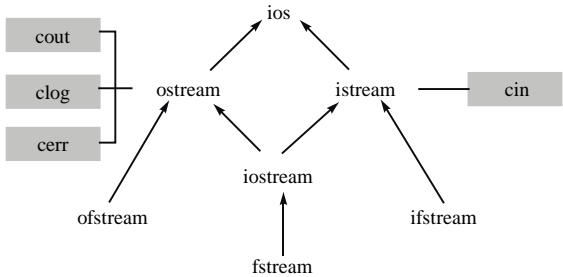


图 15-2 文件流类的关系

其中，C++ 中提供的三个文件流类的功能如下。

- `ofstream`：输出流类，用于向文件中写入内容。

- ifstream: 输入流类, 用于从文件中读出内容。
- fstream: 输入/输出流类, 用于既要读又要写的文件的操作。



在 C++ 中, 对文件的操作是通过 stream 的子类 fstream(file stream) 来实现的, 所以要用这种方式操作文件, 就必须加入头文件 fstream.h。

15.2 文件的打开与关闭

C++ 中, 要进行文件的输入/输出, 必须先创建一个流, 再把这个流与文件相关联 (即打开文件), 才能进行输入/输出操作, 完成后要关闭文件。前面提到了 C++ 中的三个输入/输出流类 ofstream、ifstream 和 fstream, 它们同属于 ios 类, 可访问在 ios 类中定义的所有操作。与此相对应, 为了执行文件的输入/输出操作, C++ 还提供了三个输入/输出流, 即输入流、输出流和输入/输出流。建立流就是定义流类的对象, 例如:

```
ofstream out;
ifstream in;
fstream inout;
```

建立了流以后, 就可以把某一个流与文件建立联系, 从而进行文件的读写操作了。

15.2.1 打开文件

打开文件, 就是用函数 open() 把某一个流与文件建立联系。open() 函数是上述三个流类的成员函数, 定义在 fstream.h 头文件中, 它的原型为:

```
void open(const unsigned char *, int mode, int access=filebuf::openprot);
```

其中, 第一个参数为字符串类型, 它用来传递文件名; 第二个参数为整型数据类型, 其值决定文件打开的方式; 第三个参数的值决定文件的访问方式及文件的类别, 默认方式是 filebuf::openprot。在 DOS/Windows 环境中, access 的值分别对应 DOS/Windows 的文件属性代码如下:

```
0  普通文件
1  只读文件
2  隐含文件
3  系统文件
8  备份文件
```

【范例 15-1】创建文件。该范例实现在当前文件夹下创建一个文本文件, 其中包含一个汉字字符串, 实现代码如代码清单 15-1 所示。

代码清单 15-1

```
1  #include <fstream.h>
2  void main()
3  {
4      ofstream SaveFile("file_create.txt");           //创建文件file_create.txt
5      SaveFile << "Hello World!Welcome to 21 C++";     //写入文件内容
6  }
```

【运行结果】在 Visual C++ 中新建一个【C++ Source File】文件, 在其中输入如上的代码, 编译无误后运行, 其结果如图 15-3 所示。

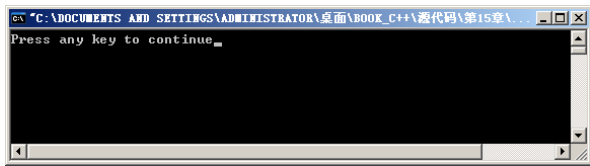


图 15-3 运行结果

读者可以看到，运行后 C++编译器并没有给出任何操作结果，这是因为操作结果没有在用户屏幕输出。读者可以打开当前文件夹，如图 15-4 所示。上述程序已经创建了文件“file_create.txt”，此外，上述程序还给该文件写入了字符串，打开该文本文件，其结果如图 15-5 所示。

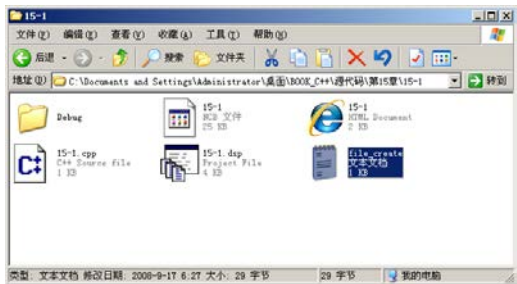


图 15-4 创建文件

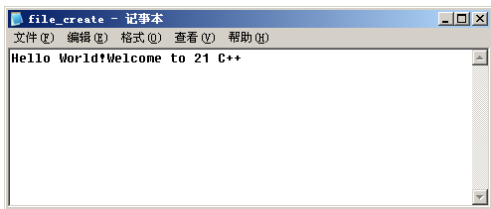


图 15-5 文本中字符串

【范例解析】第 4 行语句中，ofstream 关键字将建立一个句柄（handle），以便以后能以一个文件流的形式写入文件。SaveFile 为一个句柄名字，可换作任意标识符，“file_create.txt”为创建文件的名称。简单来说，ofstream 是一个类，ofstream SaveFile(“cpp-home.txt”)；这一语句将创建一个该类的对象；而在括号中所传递的参数实际上将传给构造函数。此处，将用户要建立的文件的名称作为实际参数传递给了该类的构造函数。



注 一旦包含了头文件 fstream.h，就不再需要（为了使用 cout/cin）包含 iostream.h，因为 fstream.h 已经自动包含了它。

通过上述范例读者可以看出，虽然已经创建了一个文件，但是并没有将其打开，如果需要对文件进行操作，首先就必须打开文件。

【范例 15-2】打开文件。该范例首先创建一个文件，接着打开该文件，如果文件不存在，则返回错误信息，否则提示文件已打开的信息，实现代码如代码清单 15-2 所示。

代码清单 15-2

```

1  #include<fstream.h>
2  void main()
3  {
4      ofstream ou("file_create.txt");           //创建输出流对象
5      ifstream in;                             //创建输入流对象
6      ou<< "Hello World!Welcome to 21 C++";    //写入文件
7      in.open("file_create.txt",ios::nocreate); //打开文件
8      if (in.fail())                           //打开失败
9          cout<<"文件不存在，打开失败！"<<endl;
10     else                                       //打开成功
11         cout<<"文件已打开，可以进行读写操作"<<endl;
12     in.close();                             //关闭文件
13 }
```

【运行结果】在 Visual C++ 中运行上述程序，其运行结果如图 15-6 所示。

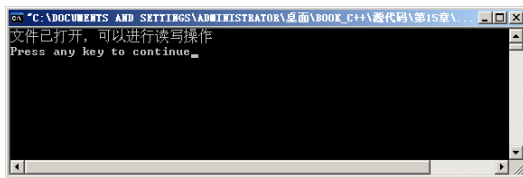


图 15-6 打开文件

【范例解析】上述代码中，首先定义 `ofstream` 对象，创建一个文件 `file_create.txt`，接着使用了 `open` 函数打开该文件，如果文件不存在，则返回错误信息，否则返回打开成功信息。其中第 7 行代码打开的文件中，参数 `ios::nocreate` 表示文件若不存在则打开失败。

15.2.2 关闭文件

文件使用后，必须将其关闭，否则可能导致数据的丢失。关闭文件就是将文件与流的联系断开，关闭文件用函数 `close()` 完成，其也是流类中的成员函数，没有参数，没有返回值，其参数原型为：

对象名.`close()`

【范例 15-3】文件的关闭。该范例使用 `open` 函数打开一个文件，并判断其是否打开，但不管其是否打开成功，在程序结束时均需要关闭文件，实现代码如代码清单 15-3 所示。

代码清单 15-3

```

1  #include<fstream.h>                                //包含头文件
2  void main()
3  {
4      ifstream in;
5      in.open("file_create.txt",ios::nocreate);        //打开文件
6      if (in.fail())
7          cout<<"文件不存在, 打开失败!"<<endl;
8      else
9          cout<<"文件已打开, 可以进行读写操作"<<endl;
10     in.close();                                       //关闭文件
11     cout<<"文件已关闭"<<endl;
12 }
```

【运行结果】在 Visual C++ 中运行上述程序，其运行结果如图 15-7 所示。

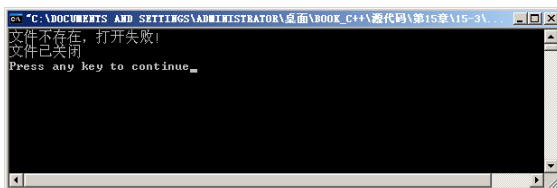


图 15-7 文件的关闭

【范例解析】上述代码中的函数 `fail()` 是流类中的成员函数，当文件以 `ios::nocreate` 方式打开时，可用该函数测试文件是否存在。若存在，返回 0，否则返回非 0。其他一些成员函数，请参考有关文献。



警告 说明打开文件的路径时，反斜杠要双写，因为编译器认为反斜杠是转义字符标志。如“C:\myfile”表示 C 盘下的 myfile 文件。



此外，在程序语句中，可将定义流与打开文件用一条语句完成，如：

```
ofstream out("test",ios::out,0);
fstream io("test",ios::in|ios::out,0);
```

一般情况下，`ifstream` 和 `ofstream` 流类的析构函数就可以自动关闭已打开的文件，但若需要使用同一个流对象打开的文件，则需要首先用 `close()` 函数关闭当前文件。

15.3 文件的顺序读写

15.2 节介绍了文件的打开和关闭，本节将重点讲解不同文件类型的文件的读写操作。在含有文件操作的程序中，必须包含头文件 `fstream.h`。

15.3.1 读写文本文件

对文本文件进行读写时，先要以某种方式打开文件，然后使用运算符“<<”和“>>”进行操作就行了，只是必须将运算符“<<”和“>>”前的 `cin` 和 `cout` 用与文件相关联的流代替。例如，`cin` 可以用 `fin` 流代替，`cout` 可以用 `fout` 代替等。

【范例 15-4】 文本文件的文件读写。该范例首先向文本文件 `test.txt` 中写入三行字符，再将该文件打开，并将写入的字符串依次输出到屏幕上，实现代码如代码清单 15-4 所示。

代码清单 15-4

```
1  #include<fstream.h>                                //包含头文件
2  int main()
3  {
4      ofstream fout("test.txt");                      //创建一个写入文件对象
5      if(!fout)                                       //创建失败
6      {
7          cout<<"不能打开输出文件。"<<endl;
8          return 1;
9      }
10     fout<<"HelloWorld! "<<endl;                    //在文件 test.txt 中输出字符串
11     fout<<10<<endl;                                //在文件 test.txt 中输出十进制数 10
12     fout<<hex<<10<<endl;                            //在文件 test.txt 中输出十六进制数
13     fout.close();                                    //关闭文件
14     ifstream fin("test.txt");                        //创建文件读取文件对象
15     if(!fin)                                         //打开失败
16     {
17         cout<<"不能打开输入文件。"<<endl;
18         return 1;
19     }
20     int i;                                          //定义变量存储文件的对应字符串
21     char ch;
22     char c[20];
23     fin>>c;                                         //读取内容放入变量中
24     fin>>i;
25     fin>>ch;
26     cout<<c<<endl;                                  //在屏幕上输出变量的值
27     cout<<i<<endl;
28     cout<<ch<<endl;
29     fin.close();                                    //关闭文件
30     return 0;
31 }
```

【运行结果】 在 Visual C++ 中运行上述程序，其运行结果如图 15-8 所示。

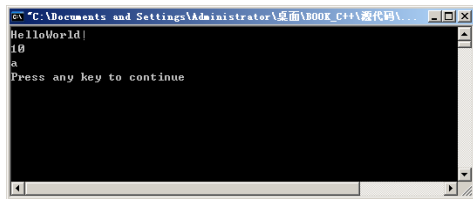


图 15-8 文本文件的读写

【范例解析】上述代码实现了文本文件的读写操作，它向文件写入若干行的字符串及数字后，通过将这些内容读入变量的方式将其输出在屏幕上。其中，输出采用流对象 `fout`，输入采用流对象 `fin`，在使用文件后都必须关闭文件。

注意 如果输出为内容较大的字符串，可以通过循环来实现将其所有字符都取入到字符数组中，并将其依次输出。

15.3.2 文本文件应用示例

在实际的程序中，文本文件的应用是较为广泛的，下面通过一个实际的范例来系统地讲解如何对文本文件进行操作。

【范例 15-5】文本文件应用示例。该范例实现对一个文本文件的多种操作，包括求文件长度、统计单词等功能，其实现代码如代码清单 15-5 所示。

代码清单 15-5

```

1  #include<fstream.h>
2  void open(char str[])           //打开文件函数
3  {
4      int i=0;
5      ifstream f;                 //创建输入流对象
6      f.open("ok.txt",ios::in);   //打开文件
7      if(!f)
8      {
9          cout<<"not open"<<endl;
10         return;
11     }
12     while(f)                     //取字符
13     {
14         f.get(str[i]);
15         i++;
16     }
17     str[i]='\0';                 //最后一个字符为结束符
18     f.close();                   //关闭文件
19     cout<<str<<endl;             //输出字符串
20 }
21 int len(char str[])              //求字符串长度函数
22 {
23     for(int i=0,temp=0;str[i]!='\0';i++) //到文件终结符为止
24         temp++;
25     return temp;
26 }
27 int index(char a[],char b[])     //找子串函数：返回子串首次出现的位置
28 {
29     int i,j,temp;
30     for(i=0;i<len(a)-len(b);i++) //循环查找

```



```

31     {
32         temp=i;
33         j=0;
34         while(j<=len(b) && a[temp]==b[j]) //逐个字符判断是否子串
35         {
36             temp++;
37             j++;
38         }
39         if(j==len(b)) //找到
40             return i;
41     }
42     return -1;
43 }
44 int count(char str[]) //统计单词个数函数
45 {
46     int i,c=0;
47     for(i=0;i<len(str);i++)
48         if(str[i]==' ')
49             c++;
50     return c+1; //比如三个单词间有两个空格
51 }
52 void output(char a[]) //输出当前文本函数
53 {
54     for(int i=0;i<len(a);i++)
55         cout<<a[i];
56     cout<<endl;
57 }
58 void output(char str[],int) //输出第一个单词
59 {
60     for(int i=0;i<len(str);i++)
61         if(str[i]==' ') //有空格则完成
62             break;
63         else
64             cout<<str[i];
65     cout<<endl;
66 }
67 void main()
68 {
69     char m[100]; //定义文件名
70     char n[]="ok"; //定义需查找的子字符串
71     open(m); //打开文件
72     cout<<"单词长度:"<<len(m)-1<<endl;
73     cout<<"目标单词首次出现的位置:"<<index(m,n)<<endl;
74     cout<<"单词数:"<<count(m)<<endl;
75     output(m); //输出当前文本
76     output(m,1); //输出第一个单词
77 }

```

【运行结果】在 Visual C++ 中运行上述程序，其运行结果如图 15-9 所示。

图 15-9 文本文件应用

【范例解析】上述代码中，通过函数的形式定义了多个文本文件操作的应用，其中包括打开文件、求文件长度、求字符串首次出现位置、统计单词个数、输出当前文本和第一个单词等函数。在主函数 `main()` 中调用这些函数即可完成对文本文件的常用操作。



提示 上述函数的定义中，涉及字符串的操作基本都是通过循环和字符数组来完成的，其中以“\0”字符来判断文件是否结束，以 `get()` 函数取字符，下面将介绍这些函数和功能。

15.3.3 二进制文件概述

前面提到了，二进制文件是一种不能用普通的字处理软件进行编辑、占空间较小的文件。二进制文件与文本文件的区别在于：

- 文本文件是字符流，二进制文件是字节流。
- 文本文件在输入时，将回车和换行两个字符转换为字符“\n”，输出是将字符“\n”转换为回车和换行两个字符，二进制文件不做这种转换。
- 文本文件遇到文件结束符时，用 `get()` 函数返回一个文件结束标志 `EOF`，该标志的值为 -1。二进制文件用成员函数 `eof()` 判断文件是否结束。
- 当文件到达末尾时，其返回一个非零值，否则返回零。当从键盘输入字符时，结束符为 `ctrl_z`，也就是说，按下 `ctrl_z`，`eof()` 函数返回的值为真。

其中，`eof()` 函数的原型如下：

```
int eof();
```

15.3.4 读写二进制文件

任何文件，都能以文本方式或二进制方式打开。对以二进制方式打开的文件，由两种方式进行读写操作：一种是使用函数 `get()` 和 `put()`，另一种是使用函数 `read()` 和 `write()`。

1. 使用函数 `get()` 和 `put()` 读写二进制文件

`get()` 函数是输入流类 `istream` 中定义的成员函数，作用是从与流对象连接的文件中读出数据，其原型为（它有许多格式，此处只介绍最一般的格式）：

```
istream &get(unsigned char &ch);
```

函数 `get()` 实现的功能为：从流中每读出一个字节或一个字符放入引用 `ch` 中返回一个流输入对象值。

`put()` 函数是输出流类 `ostream` 中定义的成员函数，作用是向与流对象连接的文件中写入数据，其原型为（它有许多格式，这里只介绍最一般的格式）：

```
istream &put(char ch);
```

函数 `get()` 实现的功能为：每将一个字节或一个字符写入流中，同样返回一个流输入对象值。



提示 `get()` 函数和 `put()` 函数都只能对单个字符或单个字节进行操作，如果实现多字符的操作，可通过循环语句来实现。

【范例 15-6】用函数 `get()` 和 `put()` 读写二进制文件。该范例定义一个命令，在 DOS 下调用该命令可实现将文件 1 的内容复制到文件 2 中，相当于 DOS 命令中的 `copy` 命令，实现代码如代码清单 15-6 所示。



代码清单 15-6

```

1  #include<fstream.h>                                //包含头文件
2  main(int argc,char *argv[])                        //带参数的 main()函数
3  {
4      char ch;
5      if (argc !=3)                                  //参数个数错误
6      {
7          cout<<"命令行输入错误! "<<endl;
8          return 1;
9      }
10     ifstream fin(argv[1]);                          //定义输入流对象, 打开第二个参数中的文件
11     if(!fin)
12     {
13         cout<<"不能打开源文件。"<<endl;
14         return 1;
15     }
16     ofstream fout(argv[2]);                         //打开第三个参数中的文件
17     if(!fout)                                       //打开失败
18     {
19         cout<<"不能打开目标文件。"<<endl;
20         return 1;
21     }
22     while(fin)                                     //循环写入
23     {
24         fin.get(ch);                                //从源文件中读出字符
25         fout.put(ch);                               //写入到目标文件中
26     }
27     fin.close();                                   //关闭流
28     fout.close();
29     return 0;
30 }

```

【运行结果】由于该程序的主函数是带参数的 main()函数, 因此可在命令提示符中运行该程序。如果在当前目录中已经有文件 file1.txt, 那么可以使用上述程序将 file.txt 中的内容复制到文件 file2.txt 中, 其运行结果如图 15-10 所示。



图 15-10 用函数 get()和 put()读写二进制文件



注意 图 15-10 中, 首先必须通过 DOS 命令 cd 进入到当前文件夹下才能调用已编译过的应用程序 15-6.exe, 再运行命令 15-6 file1.txt file2.txt 即可。读者打开当前文件夹, 可发现文件 file1.txt 和 file2.txt, 打开发现其内容均一致。

【范例解析】上述程序中, 第 10 行代码和第 16 行代码分别定义了输入流对象 fin 和输出流对象 fout, 此外, 在第 22~26 行代码中通过循环语句实现依次取出文件 file1 中的字符, 将其逐个写入到文件 file2 中, 最终实现赋值的功能。

由于上述程序使用了带参数的 `main()` 函数, 它需要参数才能运行, 因此若直接运行上述程序, 将导致执行错误, 如图 15-11 所示。

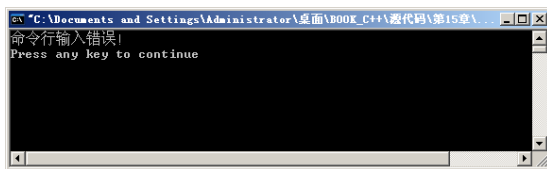


图 15-11 错误的程序执行方法

2. 使用函数 `read()` 和 `write()` 读写二进制文件

除了上述内容中讲解的使用函数 `get()` 和 `put()` 读写二进制文件外, C++ 中还支持使用函数 `read()` 和 `write()` 读写二进制文件。

`read()` 函数是输入流类 `istream` 中定义的成员函数, 其最常用的原型为:

```
istream &read(unsigned char *buf, int num);
```

该函数的作用是从相应的流中读出 `num` 字节或字符的数据, 把它们放入指针所指向的缓冲区中。第一个参数 `buf` 是一个指向读入数据存放空间的指针, 它是读入数据的起始地址; 第二个参数 `num` 是一个整数值, 该值说明要读入数据的字节或字符数。该函数的调用格式为:

```
read(缓冲区首地址, 读入的字节数);
```

需要读者注意的是, “缓冲区首地址”的数据类型为 `unsigned char *`, 当输入其他类型数据时, 必须进行类型转换。

`write()` 函数是输出流类 `ostream` 中定义的成员函数, 其最常用的原型为:

```
ostream &write(const unsigned char *buf, int num);
```

该函数的作用是从 `buf` 所指向的缓冲区把 `num` 字节的数据写到相应的流中。其参数的含义、调用及注意事项与函数 `read()` 相同。

【范例 15-7】用函数 `read()` 和 `write()` 读写二进制文件。该范例向文件 `test.txt` 写入一个双精度数据类型的数值和一个字符串, 并将该文件中的内容读出显示到屏幕中, 实现代码如代码清单 15-7 所示。

代码清单 15-7

```
1  #include<fstream.h>                                //包含头文件
2  #include<string.h>
3  main()
4  {
5      ofstream outf("test.txt");                      //定义对象, 打开文件
6      if (!outf)                                       //打开失败
7      {
8          cout<<"不能打开输出文件!"<<endl;
9          return 1;
10     }
11     double n=123.44;                                //定义需写入的变量
12     char str[]="向文件写入双精度数和字符串";
13     outf.write((char *)&n,sizeof(double));
14     outf.write(str,strlen(str));                    //调用写入函数
15     outf.close();                                    //关闭文件
16     ifstream inf("test.txt");                        //定义对象, 打开文件
17     if (!inf)
18     {
```



```

19         cout<<"不能打开输入文件！"<<endl;
20         return 1;
21     }
22     inf.read((char *)&n,sizeof(double));           //读取文件其中的内容
23     inf.read(str,26);                               //存入变量中
24     cout<<n<<"    "<<str<<endl;                   //输出变量中的值
25     inf.close();                                     //关闭文件
26     return 0;
27 }

```

【运行结果】在 Visual C++ 中运行上述程序，其运行结果如图 15-12 所示。



图 15-12 用函数 read() 和 write() 读写二进制文件

【范例解析】上述代码中，第 5 行代码定义了一个输出流对象 outf，第 22 行代码定义了一个输入流对象 inf。程序首先通过函数 write() 向文件 test.txt 写入一个双精度数据和一个字符串，接着通过函数 read() 将文件中的数值和字符串输出。

提示 读者同时也可打开当前文件夹，在其中会发现文本文件 test.txt，打开该文件可看到图 15-12 中的双精度数据和字符串。

需要说明的是，上述四个函数 get()、put()、read() 和 write() 也可以用于文本文件，其处理过程与二进制文件的处理过程基本相同，这里不再介绍。

15.4 文件的随机读写

前面介绍的有关文件的读写操作，都是按一定的顺序进行读写的，称为顺序文件，其特点是只能按数据在文件中的排列顺序一个一个地访问数据，使用很不方便。为此，C++ 又提供了文件的随机读写。

注意 随机读写是通过使用输入或输出流中与随机移动文件指针相关的成员函数，通过随意移动文件指针而达到随机访问。

移动文件指针的成员函数主要有 seekg() 和 seekp()，它们的常用原型为：

```

isream &seekg(streamoff offset, seek_dir origin);
osream &seekp(streamoff offset, seek_dir origin);

```

其中，参数 origin 表示文件指针的起始位置，offset 表示相对于这个起始位置的位移量。seek_dir 是系统定义的枚举名，origin 是枚举变量。origin 的取值有三种情况：

- ios::beg—从文件头开始，把文件指针移动由 offset 指定的距离。
- ios::cur—从文件当前位置开始，把文件指针移动由 offset 指定的距离。
- ios::end—从文件尾开始，把文件指针移动由 offset 指定的距离。

显然，当 origin 的值为 ios::beg 时，offset 的值为正；当 origin 的值为 ios::cur 时，offset 的值为负；当 origin 的值为 ios::end 时，offset 的值可正可负。为正数时从前向后移动文件指针，为负数时从后向前移动文件指针。位移量 offset 的类型是 streamoff，该类型为一个 long 型数据，

它在头文件 `iostream.h` 中定义如下:

```
typedef streamoff long;
```

此外, 移动文件指针的成员函数 `seekg()` 和 `seekp()` 的使用范围和功能为:

- 函数 `seekg()` 用于输入文件, 将文件的读指针从 `origin` 说明的位置移动 `offset` 字节。
- 函数 `seekp()` 用于输出文件, 将文件的写指针从 `origin` 说明的位置移动 `offset` 字节。

进行文件的随机读写时, 可用下列函数确定文件当前指针的位置:

```
streampos tellg();
streampos tellp();
```

其中, `streampos` 是在头文件 `iostream.h` 中定义的类型, 是 `long` 型的。函数 `tellg()` 用于输入文件, 函数 `tellp()` 用于输出文件。

【范例 15-8】 文件的随机读写。该范例实现文件的随机读写, 将文件中的字符串输出, 实现代码如代码清单 15-8 所示。

代码清单 15-8

```
1  #include<fstream.h>                                //包含头文件
2  #include<stdlib.h>
3  main(int argc,char *argv[])                        //带参数的main()函数
4  {
5      char ch;
6      if (argc !=3)                                  //参数个数不为3
7      {
8          cout<<"命令行输入错误! "<<endl;
9          return 1;
10     }
11     ifstream fin(argv[1]);                          //定义输入流并打开源文件
12     if (!fin)                                       //打开失败
13     {
14         cout<<"不能打开输入文件! "<<endl;
15         return 1;
16     }
17     fin.seekg(atoi(argv[2]),ios::beg);           //在源文件的从文件头开始读取参数3指定的距离
18     while(!fin.eof())
19     {
20         fin.get(ch);                                //逐个字符读取
21         cout<<ch;                                    //输出在屏幕上
22     }
23     fin.close();
24     return 0;
25 }
```

【运行结果】 同样, 由于该程序的主函数是带参数的 `main()` 函数, 因此可在命令提示符中运行该程序。进入当前文件夹下, 输入 “15-7 test 3” 命令后, 它表示从第 3 个位置起开始显示, 返回结果如图 15-13 所示。

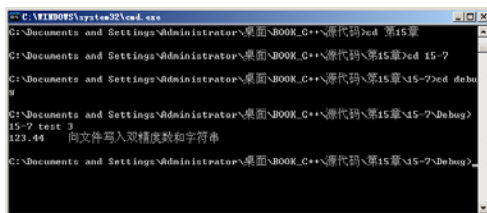


图 15-13 文件的随机读写



【范例解析】上述代码中，首先判断参数是否为三个（包括命令），在第 11 行代码中定义了输入流 `fin`，第 17 行代码使用命令 `seekg` 移动命令指针，从头开始移动用户输入的第三个参数的位数，完成后使用循环语句将结果输出。



提示 读者应该注意到了，随机文件读取和顺序文件读取的区别仅仅在于前者可以从源文件中指定的某一个地方开始读取。

15.5 小结

本章主要介绍了 C++ 中文件的概念及其相关的操作。文件和流是 C++ 中输入/输出中的重要组成部分，第 14 章也介绍过输入/输出流类库，本章重点讲解了文件流的操作。对于任意一个文件进行操作前，都必须打开文件，再进行文件的读写操作。本章通过实例详细介绍了文本文件、二进制文件和随机文件的读写操作，在对文件进行操作后，都必须将文件关闭。

15.6 习题

1. 写出下列函数实现的功能。

```
#include <iostream.h>
#include <fstream>
void JC(char *fname,int n)
{
    ofstream fout(fname,ios::out | ios::binary);
    int x;
    for (int I=0;I<n;I++)
    {cin>>x;
    fout.write((char *)&x,sizeof(x));
    }
    fout.close();
}
```

【解答】该习题主要考查输入/输出文件流的相关操作。上述函数首先创建了一个输出流对象 `fout`，通过循环语句接收用户从键盘的输入，调用 `write` 函数写入到文件中，写入完毕后将文件关闭。因此，该函数的功能为：逐个输入 `n` 个字符（一个字符串），以二进制的方式保存到 `fname` 所标识的文件中。

2. 编写一个 C++ 程序，将指定的一个文件内容读出，要求加入判断该文件是否打开成功的代码。

【解答】该习题主要考查文件的读操作。根据前面学习的内容，读者知道读出一个文件的内容必须先以某种方式将指定文件打开，并通过判断打开函数的返回值判断文件是否打开成功，如果打开成功则可以通过 `read` 函数读出。其简要的实现代码如下所示。

```
ifstream fin("d:\\简介.txt",ios::nocreate);
if(!fin){
    cout<<"File open error!\n";
    return;
}
char c[80];
while(!fin.eof()) //判断文件是否读结束
{
    fin.read(c,80);
    cout.write(c,fin.gcount());
}
fin.close();
```

3. 分析下面程序的功能。

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
void JD(char *fname)
{ifstream fin(fname,ios::in | ios::nocreate | ios::binary);
int x,s=0,n=0;
while(fin.read((char *)&x,sizeof(x)))
{s+=x;
n++;
}
cout<<n<<' ' <<s<<' ' <<float(s)/n<<endl;
fin.close();
}
```

【解答】该习题主要考查文件的读入操作。上述程序首先打开指定的文件 `fname`，通过循环读出该文件中的字符，每读出一个字符后变量 `s` 的值加上该字符的长度，而变量 `n` 的值累加 1，在 `cout` 语句中输出该变量 `s` 的值和表达式 `float(s)/n` 的值。因此，该函数的功能是：对 `fname` 所标识的二进制文件中的整数进行统计个数，求和，求平均值并显示出来。

4. 编写一个 C++ 程序，实现文件复制的功能。

【解答】该习题主要考查二进制文件的读写操作。对于一个文件的读写操作而言，首先需要先打开指定文件，任何打开目标文件，通过循环语句，从指定的源文件中分块读出数据信息，写入到目标文件中，一直到源文件读写完成为止，完成后将源文件和目标文件都关闭。其简要的实现代码如下所示。

```
ifstream fin("C:\\1.exe",ios::nocreate|ios::binary);
if(!fin){
    cout<<"File open error!\n";
    return;
}
ofstream fout("C:\\2.exe",ios::binary);
char c[1024];
while(!fin.eof())
{
    fin.read(c,1024);
    fout.write(c,fin.gcount());
}
fin.close();
fout.close();
```

5. 创建一个文本文件，写入数据后读取并显示在屏幕上。例如，写入 1234567890 后将在用户屏幕和文件中显示如图 15-14 和图 15-15 所示的结果。

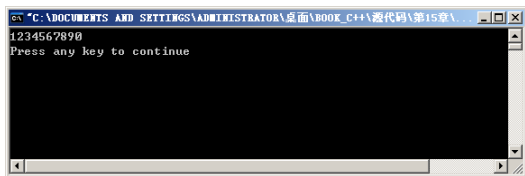


图 15-14 执行结果



图 15-15 写入到文件的数据

【解答】该程序段可将文件名和路径通过宏定义的方式给出，通过输入/输出流对象向文件 `test.txt` 写入数字 1234567890，再定义指针数组 `buf`，将文件中的数据读入该数组中，通过 `cout` 对象输出在屏幕上。其简要的实现代码如下所示。



```

#include <fstream>                                // 包含头文件
using namespace std;                              // 使用命名空间
#ifdef WIN32                                       // 预编译指令, 如果为 WIN32 系统
#define TEST_FILE "test.txt"                    // 指定文件名
#else
#define TEST_FILE "/tmp/test.txt"                // 指定文件名和路径
#endif                                           // 结束条件编译
void test()                                       // 定义函数
{
    // 首先将文件内容写为 1234567890
    {
        //fstream sfs;                          // 调试使用的注释
        //sfs.open(TEST_FILE, ios_base::out);
        fstream sfs(TEST_FILE, ios_base::out);

        char buf[] = "1234567890";               // 注意, 文件是写入方式, 会覆盖原来内容
        sfs.write(buf, sizeof(buf));             // 定义字符串并初始化
        sfs.close();                             // 写入文件
                                                // 关闭文件
    }
    // 下面读取文件内容并输出
    {
        int len;                                // 定义整型变量
        char* buf;                              // 定义字符串
        //fstream sfs;
        //sfs.open(TEST_FILE);
        fstream sfs(TEST_FILE);                 // 创建对象
        sfs.seekg (0, ios::end);                 // 定位到文件末尾
        vlen = sfs.tellg();                      // 返回地址, 也就相当于获得文件的长度
        sfs.seekg (0, ios::beg);                 // 获取完毕之后, 将文件指针提到前面
        buf = new char[len];                     // 申请一段空间
        sfs.read(buf, len);                      // 读取文件内容到 buf
        cout << buf << endl;                   // 输出在屏幕上
        delete []buf;                           // 释放空间
        sfs.close();                             // 关闭文件
    }
}
int main(int argc, char* argv[])                // 带参数的 main() 函数
{
    test();                                       // 调用 test() 函数
    return 0;                                    // 正常返回
}

```

第 16 章 命名空间

前面的章节中，有的应用程序已经使用到了命名空间。命名空间是 C++ 所独有的特征，C 语言中没有提供。简单来说，命名空间是 C++ 的一种机制，用来把单个标识符下存在大量有逻辑联系的程序实体组合到一起，此标识符作为此组群的名字。本章将详细讲解有关命名空间的相关概念和应用，以及有关作用域的相关内容。

以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为学习的参照。

- 理解命名空间的作用。
- 掌握命名空间的使用方法。
- 掌握类的作用域及 `this` 指针的应用方法。

16.1 命名空间

前面提到了，命名空间事实上就是一个含有许多标识符的空间，其中包含了许多标识符的定义，本节将介绍有关命名空间的基本概念。

16.1.1 什么是命名空间

很多初学 C++ 的人，对于 C++ 中的一些基本但不常用的概念感到模糊，命名空间（namespace）就是这样的概念。

在 C++ 中，名称（name）可以是符号常量、变量、宏、函数、结构、枚举、类和对象等。为了避免在大规模程序的设计中这些标识符的命名发生冲突，C++ 标准引入了命名空间的概念，其使用关键字 `namespace` 来定义。

简单地说，命名空间是为了解决 C++ 中的变量、函数的命名冲突而服务的。解决的办法就是将变量定义在一个不同名字的命名空间中。就如张家有电视机，李家也有同样型号的电视机，但人们能区分清楚，因为他们分属不同的家庭。同样，变量 `user_name` 既可以在命名空间 `a` 中，也可以在命名空间 `b` 中，但系统不会将其混淆，如图 16-1 所示。

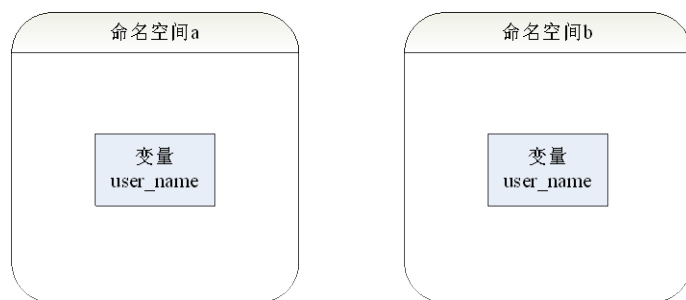


图 16-1 命名空间的概念

此外，读者还需了解几个与命名空间相关的概念。

- 声明域（declaration region）：声明标识符的区域。如在函数外面声明的全局变量，它的声明域为声明所在的文件。在函数内声明的局部变量，它的声明域为声明所在的代



码块（如整个函数体或整个复合语句）。

- 潜在作用域（potential scope）：从声明点开始，到声明域的末尾的区域。因为 C++ 采用的是先声明后使用的原则，所以在声明点之前的声明域中，标识符是不能用的。标识符的潜在作用域，一般小于其声明域。
- 作用域（scope）：即标识符对程序可见的范围。标识符在它的潜在作用域内，而并非在任何地方都是可见的。例如，局部变量可以屏蔽全局变量、嵌套层次中的内层变量可以屏蔽外层变量，从而被屏蔽的全局或外层变量在其被屏蔽的区域内是不可见的。所以，一个标识符的作用域可能小于其潜在作用域。有关作用域在后续篇章中还将做介绍。



注意 Microsoft 的 MFC（微软基础类库）中并没有使用命名空间，但是在 .NET 框架、MFC/CLI 中都大量使用了命名空间。

16.1.2 定义命名空间

C++ 中，有两种形式的命名空间：有名的命名空间和无名的命名空间。这两种命名空间的定义格式分别如下。

- 有名的命名空间：

```
namespace 命名空间名
{
    声明序列
}
```

- 无名的命名空间：

```
namespace
{
    声明序列
}
```

命名空间的成员，是在命名空间定义中的花括号内声明了的名称。读者可以在命名空间的定义内，定义命名空间的成员（称为内部定义）。也可以只在命名空间的定义内声明成员，而在命名空间的定义之外定义命名空间的成员，称为外部定义。其中，命名空间成员的外部定义的格式为：

命名空间名::成员名

【范例 16-1】定义命名空间。该范例定义了一个命名空间 Outer，其中包括变量、成员函数、子命名空间等，读者可以理解其定义语法，实现代码如代码清单 16-1 所示。

代码清单 16-1

```
1  namespace Outer                //命名空间 Outer 的定义
2  {
3      int i;                      //命名空间 Outer 的成员 i 的内部定义
4      namespace Inner            //子命名空间 Inner 的内部定义
5      {
6          void f() { i++; } //命名空间 Inner 的成员 f()的内部定义,其中的 i 为 Outer::i
7          int i;
8          void g() { i++; } //命名空间 Inner 的成员 g()的内部定义,其中的 i 为 Inner::i
9          void h();             //命名空间 Inner 的成员 h()的声明
10     }
11     void f();                  //命名空间 Outer 的成员 f()的声明
```

```

12     //namespace Inner2;           // 错误, 不能声明子命名空间
13 }
14 void Outer::f()                   //命名空间 Outer 的成员 f() 的外部定义
15 {
16     i--;
17 }
18 void Outer::Inner::h()            //命名空间 Inner 的成员 h() 的外部定义
19 {
20     i--;
21 }
22 //namespace Outer::Inner2 {/*.....*/} // 错误, 不能在外部定义子命名空间

```

【运行结果】由于命名空间必须定义在头文件中, 而不能作为一个应用程序来运行, 因此读者需要先新建一个【C++ Header file】文件。在 Visual C++ 中, 读者可以通过应用程序向导 AppWizard 来实现新建头文件, 如图 16-2 所示。在图 16-2 中新建了一个 out.h 头文件后, 就可以将代码 16-1 中的代码写入头文件中, 如图 16-3 所示。

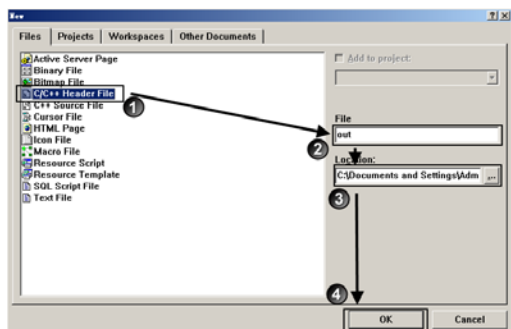


图 16-2 定义命名空间

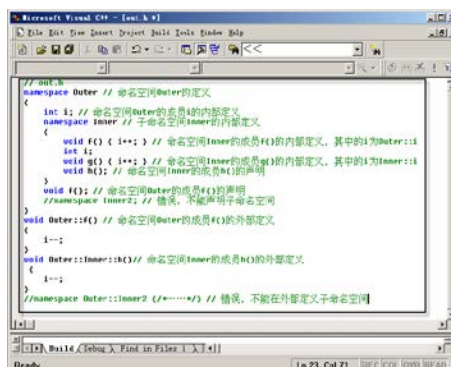


图 16-3 头文件编辑框

【范例解析】该范例中, 定义了一个命名空间 **Outer**, 其中定义了包括变量、成员函数等内容, 并且将其定义在头文件 **out.h** 中, 以便在应用程序中调用。

注意 不能在命名空间的定义中声明 (另一个嵌套的) 子命名空间, 只能在命名空间的定义中定义子命名空间。也不能直接使用“命名空间名::成员名 ……”定义方式, 为命名空间添加新成员, 而必须先要在命名空间的定义中添加新成员的声明。

此外, 命名空间是开放的, 即可以随时把新的成员名称加入到已有的命名空间中。方法是多次声明和定义同一命名空间, 每次添加自己的新成员和名称。例如:

```

namespace A
{
    int i;
    void f();
} // 现在 A 有成员 i 和 f()
namespace A {
    int j;
    void g();
} // 现在 A 有成员 i、f()、j 和 g()

```

此外, 用户还可以用多种方法, 来组合现有的命名空间, 让其为用户所用。实现方法是使用 **using** 命令包含其他的命名空间, 例如:

```

namespace My_lib
{

```



```
using namespace His_string;
using namespace Her_vector;
using Your_list::List;
void my_f(String &, List &);
}
```

16.2 使用命名空间

由于命名空间的定义中包含了许多变量及函数的定义，那么在实际的程序设计中，如何来使用命名空间中的这些定义呢？标准 C++ 给出了几种引用命名空间内变量和函数的方法，它们分别是使用作用域运算符、using 指令和 using 声明，下面将依次介绍。

16.2.1 使用作用域运算符引用成员

前面讲解过作用域运算符::，比如多个类中定义了相同的成员函数时，需要通过作用域运算符指明是属于哪个类的，而使用作用域运算符引用命名空间中的成员也是如此。例如，在前面的范例 16-1 中，创建了头文件 out.h，其中包含命名空间 Outer 的定义，同时包含变量和成员函数，下面的范例将引用该命名空间中的成员。

【范例 16-2】使用作用域运算符引用成员。该范例引用命名空间 Outer 中的成员，其实现代码如代码清单 16-2 所示。

代码清单 16-2

```
1  #include "out.h"                                // 预编译头文件
2  #include <iostream>
3  void main ( )
4  {
5      Outer::i = 0;                                // 引用命名空间中的变量 i
6      Outer::f();                                  // 引用函数，其相当于 Outer::i = -1;
7      Outer::Inner::f();                          // 相当于 Outer::i = 0;
8      Outer::Inner::i = 0;                        // 引用子命名空间中的变量 i
9      Outer::Inner::g();                          // 相当于 Inner::i = 1;
10     Outer::Inner::h();                          // 相当于 Inner::i = 0;
11     std::cout << "Hello, World!" << std::endl; // 引用命名空间 std 的函数
12     std::cout << "Outer::i = " << Outer::i << ", Inner::i = " << Outer::Inner::i
<< std::endl;
13 }
```

【运行结果】在 Visual C++ 中将范例 16-1 中定义的头文件 out.h 复制到当前文件夹下，输入上述代码，编译无误后运行，其结果如图 16-4 所示。

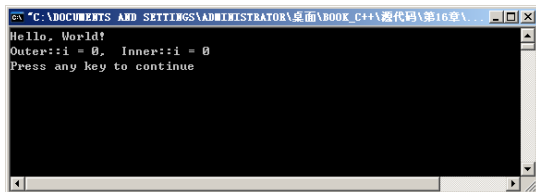


图 16-4 使用作用域运算符引用成员

【范例解析】在代码清单 16-2 中，使用了作用域运算符引用命名空间 Outer 中的成员，其中代码第 5~6 行是直接引入该命名空间中的变量和函数，代码第 7~10 行是引用命名空间 Outer 中子命名空间 Inner 中的成员。在代码第 11~12 行引入命名空间 std 中的成员函数 cout 输出部分字符和赋值的结果。

读者需要注意的是,代码清单 16-2 中的预编译部分为 `#include <iostream>`,而并不是以前常用的 `#include <iostream.h>`,这是因为在新的 C++ 标准中,生成新头文件的方法仅仅是将现有 C++ 头文件名中的 `.h` 去掉。以前的 C++ 头文件名如 `<iostream.h>` 将会继续被支持,尽管它们不在官方标准中,但这些头文件的内容不在命名空间 `std` 中。

新的 C++ 头文件如 `<iostream>` 包含的基本功能和对应的旧头文件相同,但头文件的内容在命名空间 `std` 中。因此,在代码 16-2 中可以直接调用该文件中命名空间 `std` 的函数。



提示 标准 C++ 库 (不包括标准 C 库) 中所包含的所有内容 (包括常量、变量、结构、类和函数等) 都被定义在命名空间 `std` (standard 标准) 中了,如上述代码中的 `cout` 对象。

16.2.2 使用 using 指令

读者从范例 16-2 可以看出,每次引用命名空间中的成员都必须通过作用域运算符来实现是非常烦琐的。为了省去每次调用 `Inner` 成员和标准库的函数和对象时,都要添加 `Outer::`、`Inner::` 和 `std::` 的麻烦,可以使用标准 C++ 的 `using` 编译指令来简化命名空间中的名称的使用。格式为:

```
using namespace 命名空间名[::命名空间名……];
```

在这条语句之后,就可以直接使用该命名空间中的标识符,而不必写前面的命名空间定位部分。因为 `using` 指令,使所指定的整个命名空间的成员都直接可用。

【范例 16-3】使用 `using` 指令。该范例将范例 16-2 中的代码进行改写,使用 `using` 指令,改写后的代码如代码清单 16-3 所示。

代码清单 16-3

```

1  #include "out.h"                                // 预编译头文件
2  #include <iostream>
3  // using namespace Outer;                        // 编译错误, 因为变量 i 和函数 f() 有名称冲突
4  using namespace Outer::Inner;                   // 对子命名空间使用了 using 指令
5  using namespace std;                             // 对系统空间 std 使用 using 指令
6  void main ( )
7  {
8      Outer::i = 0;                                // 引用命名空间中的变量 i
9      Outer::f();                                   // 引用函数, 其相当于 Outer::i = -1;
10     f();                                           // 可直接引用子命名空间函数
11     i = 0;                                         // 可直接引用子命名空间中的变量 i
12     g();                                           // 可直接引用子命名空间函数
13     h();                                           // 可直接引用子命名空间函数
14     cout << "Hello, World!" << std::endl;         // 可直接引用命名空间 std 的函数
15     cout << "Outer::i = " << Outer::i << ", Inner::i = " << i << endl;
16 }
```

【运行结果】同样,由于该范例仍然使用 `out.h` 头文件,因此需要将范例 16-1 中定义的头文件 `out.h` 复制到当前文件夹下。在 Visual C++ 的编译环境下,输入如上的代码,编译无误后运行,其结果如图 16-5 所示。

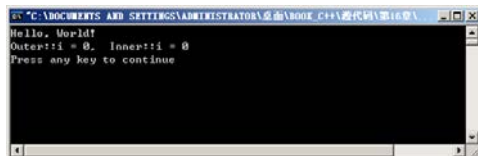


图 16-5 使用 using 指令



【范例解析】读者可以看到，上述代码的执行结果与使用作用域运算符是相同的，而由于使用了 using 指令，使得引用命名空间内的成员的写法大大简化了。



注意 上述代码只是对 Outer 命名空间的子命名空间 Inner 使用 using 指令进行预编译，在调用该子命名空间中的成员可直接调用，但对于 Outer 命名空间中的成员，还是只能通过作用域运算符的形式引用，而不能省略。

16.2.3 使用 using 声明

除了可以使用 using 编译指令（组合关键字 using namespace）外，还可以使用 using 声明来简化对命名空间中的名称的使用，其使用格式为：

```
using 命名空间名::[命名空间名::.....]成员名;
```

读者需要注意的是，此处的格式中关键字 using 后面并没有跟关键字 namespace，而且最后必须为命名空间的成员名（而在 using 编译指令的最后，必须为命名空间名）。

与 using 指令不同的是，using 声明只是把命名空间的特定成员的名称添加到所在的区域中，使得该成员可以不需要采用命名空间的作用域解析运算符来定位，而直接被使用。但是该命名空间的其他成员，仍然需要作用域运算符来定位。

【范例 16-4】使用 using 声明。该范例使用 using 声明说明了命名空间，实现代码如代码清单 16-4 所示。

代码清单 16-4

```
1  #include "out.h"                                // 预编译头文件
2  #include <iostream>
3  using namespace Outer;                          // 注意，此处无::Inner
4  using namespace std;
5  // using Inner::f;                               // 编译错误，因为函数 f() 有名称冲突
// 此处省去 Outer::，是因为 Outer 已经前面的 using 指令作用过了
6  using Inner::g;
7  using Inner::h;
8  void main ( )
9  {
10     i = 0;                                       // 相当于 Outer::i=0
11     f();                                       // 引用函数，其相当于 Outer::i = -1;
12     Inner::f();                               // 引用子命名空间函数
13     Inner::i = 0;                             // 引用子命名空间中的变量 i
14     g();                                       // 相当于 Inner::g(), Inner::i = 1;
15     h();                                       // 相当于 Inner::h(), Inner::i = 0;
16     cout << "Hello, World!" << std::endl;    // 可直接引用命名空间 std 的函数
17     cout << "Outer::i = " << Outer::i << ", Inner::i = " << i << endl;
18 }
```

【运行结果】在 Visual C++ 中将范例 16-1 中定义的头文件 out.h 复制到当前文件夹下，输入如上的代码，编译无误后运行，其结果如图 16-6 所示。

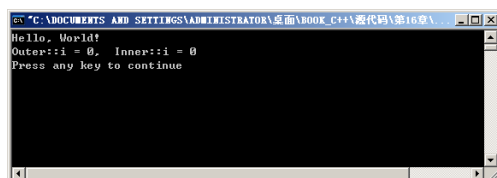


图 16-6 使用 using 声明

【范例解析】上述代码中，第 3~4 行代码使用 `using` 指令对 `Outer` 和 `std` 命名空间进行了预编译，而在第 6~7 行使用 `using` 声明了 `Inner` 命名空间内的两个成员，在第 14~15 行引用这两个成员时就不用使用作用域运算符，而可以直接引用。其他没有声明的成员则需要使用作用域运算符来引用，如函数 `f()` 和变量 `i`。

16.2.4 using 指令与 using 声明的比较

通过上述 `using` 指令和 `using` 声明的学习，读者可以看到，`using` 编译指令和 `using` 声明都可以简化对命名空间中名称的访问。

`using` 指令使用后，可以一劳永逸，它对整个命名空间的所有成员都有效，非常方便。而 `using` 声明，则必须对命名空间的不同成员名称逐个地去声明，非常麻烦。

但是，一般来说，使用 `using` 声明会更安全。因为，`using` 声明只会导入指定的名称，如果该名称与局部名称发生冲突，编译器会报错。而 `using` 指令导入整个命名空间中的所有成员的名称，包括那些可能根本用不到的名称，如果其中有名称与局部名称发生冲突，则编译器不会发出任何警告信息，而只是用局部名称去自动覆盖命名空间中的同名成员。

虽然使用命名空间的方法有多种可供选择。但是不能贪图方便而一味地使用 `using` 指令，这样就完全背离了设计命名空间的初衷，也失去了命名空间应该具有的防止名称冲突的功能。



提示 一般情况下，对偶尔使用的命名空间成员，应该使用命名空间的作用域解析运算符来直接给名称定位。而对一个大命名空间中的经常要使用的少数几个成员，提倡使用 `using` 声明，而不应该使用 `using` 编译指令。只有需要反复使用同一个命名空间的多数成员时使用 `using` 编译指令，才被认为是可取的。

例如，如果一个程序只使用一两次 `cout`，而且也不使用 `std` 命名空间中的其他成员，则可以使用命名空间的作用域解析运算符来直接定位。如代码清单 16-4 中：

```
#include <iostream>
.....
std::cout << "Hello, World!" << std::endl;
std::cout << "Outer::i = " << Outer::i << ", Inner::i = " << Outer::Inner::i <<
std::endl;
```

又如，如果一个程序要反复使用 `std` 命名空间中的 `cin`、`cout` 和 `cerr` 等函数，而不怎么使用 `std` 命名空间中的其他成员，则应该使用 `using` 声明而不是 `using` 指令。如：

```
#include <iostream>
.....
using std::cout;
cout << "Hello, World!" << endl;
cout << "Outer::i = " << Outer::i << ", Inner::i = " << Outer::Inner::i << endl;
```

16.3 类的作用域

类的作用域简称类域，它是指在类的定义中由一对花括号所括起来的部分。每一个类都具有该类的类域，该类的成员均在该类所属的类域中。

16.3.1 静态数据成员

在类的定义中可知，类域中可以定义变量，也可以定义函数。此外，在类域中的静态成员和成员函数还具有外部的连接属性。

简单来说，C++ 中的静态成员是指在类中通过关键字 `static` 说明的成员，其主要包括静态



数据成员和静态成员函数。静态成员主要用于解决同一个类的不同对象之间数据和函数共享的问题，同一个类的不同对象的静态成员均使用同一个内存空间。

静态数据成员是指类中用关键字 `static` 说明的那些数据成员。它是类的数据成员的特例，每个类只有一个静态数据成员的拷贝，从而实现同类对象之间的数据共享。

【范例 16-5】该范例调用了静态数据成员，实现代码如代码清单 16-5 所示。

代码清单 16-5

```
1  #include <iostream.h>
2  class test                               //定义类
3  {
4      int k;                               //定义私有成员
5  public:
6      static int n;                         //静态数据成员
7      test(int kk)                         //定义构造函数
8      {
9          k=kk;                             //类内直接引用静态数据成员
10         n++;
11     }
12     void disp()                           //定义成员函数
13     {
14         cout<<"n="<<n<<"    k="<<k<<endl;
15     }
16 };
17 int test::n=0;                           //类外对静态数据成员初始化
18 void main()
19 {
20     test t1(10);                           //创建对象
21     t1.disp();                             //调用成员函数
22     test t2(20);                           //创建对象
23     t2.disp();
24     test::n++;                             //调用静态数据成员
25     t2.disp();
26 }
```

【运行结果】在 Visual C++中执行上述代码，其结果如图 16-7 所示。

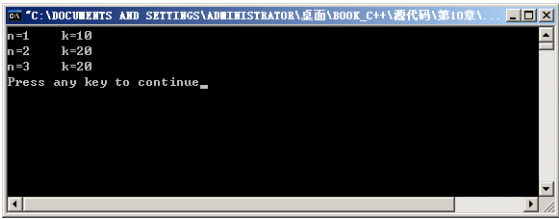


图 16-7 静态数据成员

【范例解析】上述代码中，在类 `test` 中声明了静态数据成员 `n`（其为公有成员变量），并在类外对该静态数据成员进行了初始化。在主函数 `main()`中，由 `test` 类创建了三个对象 `t1`、`t2` 和 `t3`，分别调用了构造函数对其数据成员进行初始化，其中使用了静态数据成员 `n` 进行递加运算，使用方法与普通变量的使用相同。

提 在函数中调用静态数据成员，一定要使用作用域运算符指明该数据成员所属的类，如上述代码中的第 24 行。

16.3.2 静态成员函数

静态成员函数可以直接引用该类的静态数据成员和成员函数，但不能直接引用非静态数据成员。如果要引用非静态数据成员，必须通过参数传递的方式得到对象名，再通过对象名来引用。

使用静态成员函数时应注意以下问题：

- 静态成员函数可以在类内定义，也可以在类外定义，此时，不要用前缀 `static`。
- 系统限定静态成员函数为内部连接，这样就不会因与文件连接的其他同名函数相冲突，保证了静态成员函数的安全性。另外，可以用静态成员函数在建立任何对象之间去处理静态数据成员，这是普通成员函数办不到的。
- 静态成员函数中没有隐含 `this` 指针，有关 `this` 指针将在后续章节中详细介绍。静态成员函数的调用时可用下面两种方法之一。

<类名>::<静态函数名(>;

或

<对象名>::<静态函数名(>;

- 静态成员函数不能访问类中的非静态成员。若要访问，只能通过对象名或指向对象的指针来访问。

【范例 16-6】静态成员函数的使用。该范例调用了静态成员函数，实现代码如代码清单 16-6 所示。

代码清单 16-6

```

1  #include <iostream.h>
2  class point                                //定义类
3  {
4      static int t;                          //定义静态数据成员
5      int a,b;
6  public:
7      point(int aa=0,int bb=0)              //定义构造函数
8      {
9          a=aa;
10         b=bb;
11         t++;
12     }
13     point(point &p);                       //重载构造函数
14     int geta()                             //定义成员函数
15     {
16         return a;
17     }
18     int getb()
19     {
20         return b;
21     }
22     static void getc()                     //静态成员函数
23     {
24         cout<<"object id:"<<t<<endl;
25     }
26 };
27 point::point(point &p)                    //定义重载构造函数
28 {
29     a=p.a;
30     b=p.b;
31     t++;
32 }
```



```

33  int point::t=0;                                //类外对静态数据成员初始化
34  void main()
35  {
36      point getc();                                //用类名引用静态成员函数
37      point ab(2,3);
38      cout<<"point ab:"<<ab.geta()<<" "<<ab.getb()<<endl;
39      ab.getc();                                    //用对象名引用静态成员函数
40      point ba(ab);
41      cout<<"point ba:"<<ba.geta()<<" "<<ba.getb()<<endl;
42      point::getc();                                //用类名引用静态成员函数
43  }

```

【运行结果】在 Visual C++ 中执行上述代码，其结果如图 16-8 所示。

【范例解析】上述代码中定义了类 point，其中包含了静态成员函数 getc() 的定义，该成员函数为公有成员。此外，还定义了私有静态成员变量 t。在主函数 main() 中，分别通过类名定义、对象名和类名引用该静态成员函数，实现输出的功能。

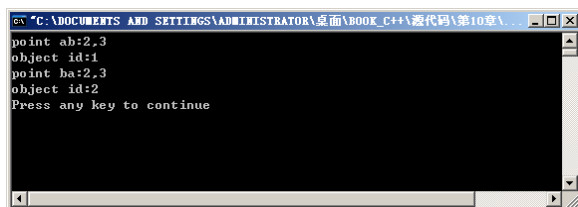


图 16-8 静态成员函数



作为成员函数，静态数据成员的访问受到类的严格控制。对于公有的静态成员函数，可以通过类名或对象名来调用；而对于普通的静态成员函数，只能通过对象名来调用。

16.4 作用域

前面提到的作用域是指类或变量的作用范围。事实上，C++ 程序语言的标识符作用域有三种：全局作用域、局部作用域、文件作用域。

简单地说，域就是范围，而作用应理解为起作用，也可称为有效。所以作用域就是指一个变量或函数在代码中起作用的范围，或者说，一个变量或函数的“有效范围”。就如枪发出的子弹，有一定的射程，超出了这个射程，就是超出了子弹的有效范围，这颗子弹就失去了作用。代码中的变量或函数，有的可以在整个程序中的所有范围内起作用，这称为“全局”的变量或函数。而有的只能在一定的范围内起作用，称为“局部”变量。

16.4.1 局部作用域

一对 {} 括起来的代码范围，属于一个局部作用域。如果这个局部作用域包含更小的子作用域，那么子作用域具有较高的优先级。在一个局部作用域内，变量或函数从其声明或定义的位置开始，一直作用到该作用域结束为止。

【范例 16-7】定义局部变量，说明该变量的作用域，实现代码如代码清单 16-7 所示。

代码清单 16-7

```

1  #include <iostream>
2  using std::cout;                                //指明命名空间
3  using std::endl;
4  void func()                                       //定义函数

```

```

5  {
6      int a;
7      a = 100;
8      cout << a << endl;           //输出 a 的值
9  }
10 int main(int argc, char* argv[]) //带参数的 main() 函数
11 {
12     cout << a << endl;           // 错误: 变量 a 未定义
13     return 0;
14 }

```

【运行结果】在 Visual C++ 中新建一个【C++ Source File】文件，输入如上的代码，使用快捷键【Ctrl+F7】进行编译时，系统给出如图 16-9 所示的错误信息。

【范例解析】上述代码中，在函数 func() 中定义了变量 a，但这个变量的“作用域”在}之前停止。所以，出了花括号以后，变量 a 就不存在了。因此，在主程序 main() 中，编译系统认为变量 a 是没有定义的，给出如图 16-9 所示的错误提示。

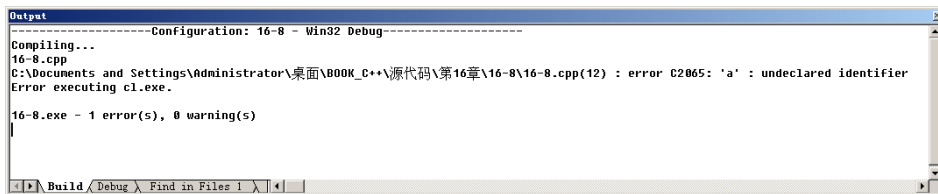


图 16-9 错误提示

提示 上述程序中没有使用<iostream.h>头文件，而是用 using 声明了在程序中使用到的 cout 函数和 endl。

局部作用域内定义的变量，其有效范围从它定义的行开始，一直到该局部作用域结束。在局部作用域内定义的变量，称为局部变量，这在以前也介绍过。范例 16-7 中的局部作用域是一个函数。变量只在其作用域内有效，即在一对{}中有效。

范例 16-7 中的局部作用域是一个函数，事实上，除了在上述的函数中用到了{}外，在所有使用到复合语句的地方都用到了局部作用域。例如，在选择结构 if 语句中定义了变量 a，那么 a 是一个局部变量，处在 if 语句所带的那对{}之内。

```

if( i> j)
{
    int a;
    ...
}

```

又如，下面的 for 语句中也定义了变量 a，其也是一个局部变量。处在 for 语句带的{}之内起作用，之外就不再起作用了。

```

for(int i=0;i<100;i++)
{
    int a;
    ... ..
}

```

for 语句涉及局部作用域时，有一点需要特别注意：代码 16-7 中，变量 i 的作用域是什么？根据最新的 ANSI C++ 规定，在 for 的初始语句中声明的变量，其作用范围是从它定义的位置开始，一直到 for 所带语句的作用域结束。而原来的标准是出了 for 语句仍然有效，直到 for 语句外层的局部作用域结束。



此外，即使没有流程控制语句，读者也可以根据需要，在代码中直接加上一对{}，人为地制造一个局部作用域。比如在某个函数中：

```
void func()
{
    int a = 100;
    cout << a << endl;
    {
        int a = 200;
        cout << a << endl;
    }
    cout << a << endl;
}
```

注意 上述代码人为地加了一对{}制造了一个局部作用域，其并没有实际意义，但读者可自行实验观察该函数的输出，理解局部作用域在程序中起的作用。

16.4.2 全局作用域

如果一个变量声明或定义不在任何局部作用域之内，该变量称为全局变量。同样，一个函数声明不在任何局部作用域内，则该函数是全局函数。一个全局变量从它声明或定义的行起，将一起直接作用到源文件的结束。

【范例 16-8】 变量的全局作用域。该范例声明了全局变量，说明该变量的全局作用域，实现代码如代码清单 16-8 所示。

代码清单 16-8

1	#include <iostream>	//预编译头文件
2	using namespace std;	//使用 using 指令
3	int a = 100;	//定义全局变量
4	void func()	//定义函数
5	{	
6	cout <<"func 函数中的 a= " << a << endl;	//输出全局变量 a
7	}	
8	void main()	
9	{	
10	func();	//调用函数
11	cout <<"main 函数中的 a= " << a << endl;	//输出全局变量 a
12	}	

【运行结果】 在 Visual C++中新建一个【C++ Source File】文件，输入如上的代码，编译无误后运行，其结果如图 16-10 所示。

【范例解析】 上述代码中，在程序开始第 3 行代码定义了全局变量 a，该变量不属于任何函数或复合语句块{}范围内，因此其作用域为全局作用域。读者可以看到，它既可以在函数 func 中起作用，也可以在函数 main()中起作用。

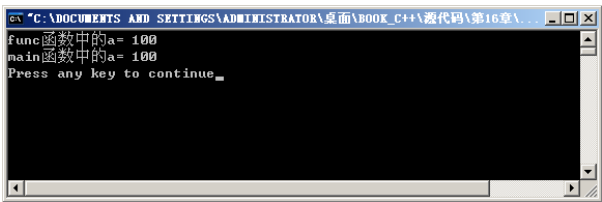


图 16-10 全局作用域



注意 全局作用域经常和作用域运算符::一起使用,作用域操作符::要求编译器将其所修饰的变量或函数看成全局的。或者说,当编译器遇到一个使用::修饰的变量或函数时,编译器仅从全局的范围内查找该变量的定义。

此外,前面介绍的命名空间也可以认为是一个作用域级别。命名空间可以是全局的,也可以位于另一个命名空间之中,但是不能位于类和代码块中。所以,在命名空间中声明的名称(标识符),默认具有外部链接特性(除非它引用了常量)。

在所有命名空间之外,还存在一个全局命名空间,它对应于文件级的声明域。因此,在命名空间机制中,原来的全局变量,现在被认为位于全局命名空间中。

16.4.3 作用域嵌套

前面介绍的作用域都是单一的作用域,即一个程序中只有全局作用域或只有局部作用域。事实上,在实际程序中经常要用到多个作用域,C++也允许在一个程序中包含既有全局作用域又有局部作用域,即作用域的嵌套。

【范例 16-9】作用域嵌套。该范例定义了全局变量和局部变量,说明这两个变量的作用域是可以嵌套的,实现代码如代码清单 16-9 所示。

代码清单 16-9

```

1  #include <iostream>
2  using std::cout;                      //使用 using 声明
3  using std::endl;
4  int a=200;                            //定义全局变量 a
5  void func()
6  {
7      int a;                            //定义局部变量 a
8      a = 100;
9      cout <<"局部变量 a= " << a << endl;    //输出局部变量 a 的值
10 }
11 int main(int argc, char* argv[])
12 {
13     func();                            //调用函数
14     cout <<"全局变量 a= " << a << endl;    //输出全局变量 a 的值
15     return 0;
16 }
```

【运行结果】在 Visual C++中新建一个【C++ Source File】文件,输入如上的代码,编译无误后运行,其结果如图 16-11 所示。

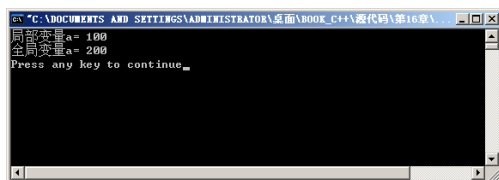


图 16-11 作用域嵌套

【范例解析】上述代码中,第 4 行代码定义了变量 a 具有全局作用域;在第 7 行定义了变量 a 在{}内,具有局部作用域,只在函数 func 中有效。在主函数 main()中,调用 func()函数输出局部变量 a 的值,并直接输出全局变量 a 的值。

上述范例中,两个变量 a,一个为全局变量,一个为局部变量。前者的作用域包含了后者



的作用域，这称为作用域的嵌套。如果在多层的作用域里，有变量同名，那么内层的变量起作用，而外层的同名变量暂时失去作用。比如在上例中，当代码执行到第 9 行时，所输出的是局部变量 a 的值。而执行到代码的第 14 行时，输出的是全局变量 a。



注意 当内层的变量和外层的变量同名时，在内层里，外层的变量暂时不可见。如果外层是全局作用域，那么可以使用::操作符来让它在内层有同名变量的情况下使用，如下代码所示：

```
int a = 0;
void func()
{
    int a;
    a = 100;
    cout << a << endl;          //输出内层的 a;
    cout << ::a << endl;        //输出全局的 a。
}
```

16.5 this 指针

在前面章节中有些代码中使用到了 this 指针，此处系统介绍该指针的作用。简单地说，this 指针是一个特殊的指针。当类的某个非静态的成员函数在执行时，this 指针指向类的一个对象，且这个对象的某个成员函数正在被调用，并作为隐含参数传递给每一个被声明的成员函数。

在实际程序中，this 指针用得最多的地方是用做返回值。使用 this 指针可以允许成员函数返回调用对象给调用者。在第 13 章运算符重载中，this 指针常作为返回值。

【范例 16-10】 this 指针的使用。该范例使用了 this 指针，其作为返回值，实现代码如代码清单 16-10 所示。

代码清单 16-10

```
1  #include <iostream>                                //声明头文件
2  #include <string.h>
3  using namespace std;                                //引用命名空间 std
4  class Date                                          //定义类
5  {
6      int mo,da,yr;
7      char *month;
8  public:
9      Date(int m=0, int d=0, int y=0);                //声明构造函数
10     ~Date();                                          //声明析构函数
11     Date& operator=(const Date&);                    //声明=运算符重载
12     void display() const;                             //声明成员函数
13 };
14 Date::Date(int m, int d, int y)                      //定义构造函数
15 {
16     static char *mos[] =                             //静态数组
17     {
18         "January", "February", "March", "April", "May", "June",
19         "July", "August", "September", "October", "November", "December"
20     };
21     mo = m; da = d; yr = y;
22     if (m != 0)
23     {
24         month = new char[strlen(mos[m-1])+1];        //创建空间
25         strcpy(month, mos[m-1]);                      //字符串复制
26     }
```

```

27     else month = 0;
28 }
29 Date::~Date() //定义析构函数
30 {
31     delete [] month; //释放空间
32 }
33 void Date::display() const //定义成员函数
34 {
35     if (month!=0) cout<<month<<' '<<da<<","<<yr<<endl; //输出显示
36 }
37 Date& Date::operator=(const Date& dt) //重载运算符=
38 {
39     if (this != &dt)
40     {
41         mo = dt.mo; //给成员变量赋值
42         da = dt.da;
43         yr = dt.yr;
44         delete [] month; //删除空间
45         if (dt.month != 0)
46         {
47             month = new char [strlen(dt.month)+1]; //定义空间
48             strcpy(month, dt.month); //字符串
49         }
50         else month = 0;
51     }
52     return *this; //返回 this 指针指向的值
53 }
54 void main()
55 {
56     Date birthday(4,19,1979); //创建对象，并调用构造函数初始化
57     Date oldday,newday; //创建对象
58     oldday=newday=birthday; //对象赋值，隐式调用运算符
59     birthday.display(); //调用成员函数
60     oldday.display();
61     newday.display();
62 }

```

【运行结果】在 Visual C++ 中新建一个【C++ Source File】文件，输入如上的代码，编译无误后运行，其结果如图 16-12 所示。

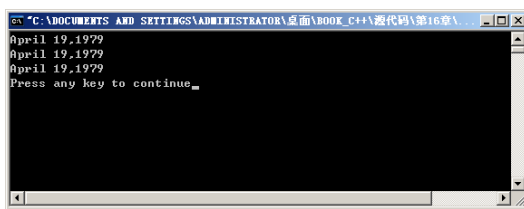


图 16-12 this 指针的应用

【范例解析】上述代码让重载赋值运算符返回了一个 Date 对象的引用，在代码的第 52 行通过 this 指针返回了一个对象的引用。读者看到，在代码的第 59~60 行中对对象调用该具有 this 指针返回值的函数时，不需带参数，也没有返回值。该范例实现的功能是重载赋值运算符，使得 Date 对象之间可以使用赋值运算符进行运算。



注意 需要读者注意的是，静态成员函数不存在 this 指针。



16.6 小结

本章主要介绍了 C++ 中命名空间的相关内容，主要包括命名空间的概念、定义，以及在具体程序中引用命名空间成员的几种方法，并将这些方法做了比较，这是本章的重点和难点。此外，本章还就作用域做了详细介绍，作用域包括变量的作用域、函数作用域和类作用域，作用域主要分为全局作用域、局部作用域和文件作用域，它们之间可以进行嵌套。最后，本章就 C++ 中的静态成员，包含静态数据成员和静态成员函数做了讲解。

16.7 习题

1. 在 C++ 程序中，为什么要引入命名空间的概念？

【解答】为了避免在大规模程序的设计中这些标识符的命名发生冲突，C++ 标准引入了命名空间的概念，其使用关键字 `namespace` 来定义。简单地说，命名空间是为解决 C++ 中的变量、函数的命名冲突而服务的。将变量定义在一个不同名字的命名空间中，即使标识符的命名相同，由于其属于不同的命名空间，也不会产生冲突。

2. 下列程序中横线处正确的语句应该是什么？

```
#include<iostream>
using namespace std;
class Base
{
public:
void fun( ){cout<< "Base : : fun" << endl;}
};
class Derived : public Base
{
public:
void fun( )
{
_____ //显式调用基类的函数 fun( )
cout << "Derived : : fun" << endl;
}
};
```

【解答】该程序段主要考查 `using` 指令和作用域运算符的应用。上述程序中定义了基类 `Base` 和派生类 `Derived`，两个类都包含公有成员函数 `fun`，在派生类中为了调用基类的 `fun` 函数，应该加上作用域运算符进行调用。因此，此处应该填写显式调用基类 `Base` 的成员函数 `fun` 的语句：`Base::fun()`。

3. 命名空间在实际的使用中，如何引用其中的成员？有哪些方法可以实现其成员的引用？各有哪些优缺点？

【解答】标准 C++ 给出了几种引用命名空间内变量和函数的方法，它们分别是使用作用域运算符、`using` 指令和 `using` 声明。一般情况下，对偶尔使用的命名空间成员，应该使用命名空间的作用域解析运算符来直接给名称定位。而对一个大命名空间中的经常要使用的少数几个成员，提倡使用 `using` 声明，而不应该使用 `using` 编译指令。只有需要反复使用同一个命名空间的多数成员时才使用 `using` 编译指令。

4. 下列程序的输出结果是什么？

```
#include<iostream.h>
void f()
{
static int i=15;
i++;
```

```

    cout<<"i="<<i<<endl;
}
void main()
{
    for(int k=0;k<2;k++)
        f();
}

```

【解答】该习题主要考查静态成员的应用。上述程序中定义了变量 *i* 为静态数据成员，其初始值为 15，在主函数中循环调用该函数 *f*。第一次调用时，输出 *i*++ 后的值为 16，第二次调用时 *i* 的值已经变为 16，此时再进行 *i*++ 运算并输出，输出 *i* 的值为 17。因此，该程序的输出结果为：*i*=16 *i*=17。

5. 编写一个实现冒泡排序的 C++ 程序，要求其中使用命名空间 *std*。

【解答】该习题主要考查冒泡排序的实现和命名空间 *std* 的应用。该程序中，需要使用命名空间 *std* 中的 *cin*、*cout* 和 *endl* 等成员，因此需使用 *using* 编译指令将其进行声明。冒泡排序的实现通过一个双重循环，内循环用于交换相邻的数组元素，外循环用于控制比较的次数。其简要的实现代码如下所示。

```

#include <iostream>
using namespace std;
int sort(int array[],int n)
{
    int temp;
    for (int i=1;i<n;i++)
    {
        for (int j=n-1;j>=1;j--)
        {
            if (array[j]<array[j-1])
            {
                temp=array[j-1];
                array[j-1]=array[j];
                array[j]=temp;
            }
        }
    }
}
int main()
{
    int array[10]={3,5,1,7,8,9,4,1,2,10};
    int i;
    cout<<"排序前: "<<endl;
    for (i=0;i<10;i++)
        cout<<array[i]<<",";
    cout<<endl;
    sort(array,10);
    cout<<"排序后: "<<endl;
    for (i=0;i<10;i++)
        cout<<array[i]<<",";
    cout<<endl;
}

```

6. 在头文件 16-11.h 中声明一个命名空间 *name*，其中包含变量和函数，在主函数中引用该命名空间内的成员，完成相应的功能。

【解答】在 Visual C++ 中，使用应用程序向导 AppWizard 新建一个【C/C++ Header file】文件，输入如下定义命名空间的代码：

```

#include <iostream>                                //注意没有加 h
using namespace std;                               //使用命名空间

```



```
static int n;                                //定义静态全局变量
namespace name                                //定义命名空间 name
{
    static void fn()                          //定义静态成员函数
    {
        static int i=10;                      //定义静态局部变量
        i = 505;                              //变量 i 赋值
        n++;                                  //变量 n 递增 1
        cout<<n<<endl;                      //输出静态全局变量的值
    }
}
```

将上述代码输入完成后保存为 16-11.h，再使用应用程序向导 AppWizard 新建一个【C++ Sourcer file】文件，输入如下代码：

```
#include <iostream>
#include "16-12.h"                            //包含头文件
using namespace std;                          //使用 using 指令
using namespace name;                        //使用命名空间
void f()                                      //定义函数
{
    static int t =808;                        //定义静态局部变量
    cout<<"t = "<<t<<endl;                  //输出值
}
void main()
{
    n=20;                                     //给静态全局变量 n 赋值
    cout<<n<<endl;                          //输出静态全局变量 n 的值
    fn();                                    //引用命名空间成员，输出 n 的值
    cout<<"t value = "<<n<<endl;            //输出值
}
```

第 17 章 引用与内存管理

引用（reference）是 C++ 引入的新语言特性，是 C++ 常用的一个重要内容之一，正确、灵活地使用引用，可以使程序简洁、高效。引用作为对象的另一名字，在实际程序中，引用主要用做函数的形式参数。简单来说，引用是一种复合类型（compound type），即用其他类型定义的类型，通过在变量名前添加“&”符号来定义。在引用的情况下，每一种引用类型都关联到某一其他类型。不能定义引用类型的引用，但可以定义任何其他类型的引用。

内存管理也是计算机程序设计中较为复杂的一个部分，本章将要重点讲解其中的动态内存管理，这可以使得 C++ 程序更加灵活多样。以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为一个学习的参照。

- 理解引用的概念。
- 掌握引用在实际程序中的使用和操作及其与指针的区别。
- 掌握动态内存分配的方法。

17.1 引用

引用是 C++ 的初学者比较容易产生迷惑的概念，引用引入了对象的一个同义词，引用只是绑定的对象的另一名字，作用在引用上的所有操作事实上都是作用在该引用绑定的对象上。

17.1.1 引用概述

简单地说，引用就是某一变量（目标）的一个别名，对引用的操作与对变量直接操作完全一样。如果对引用进行输出、赋值等操作，其执行如图 17-1 所示。

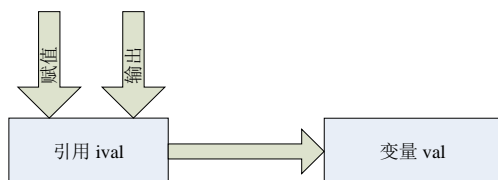


图 17-1 引用

通过图 17-1 读者可以看出，引用其实是变量的另一个名字。因此，与变量同样，在使用引用前也需要对其进行声明，引用的声明语句如下：

类型标识符 &引用名=目标变量名；

其中，参数说明如下。

- 类型标识符是指目标变量的类型。
- &在此不是求地址运算，而是起标识作用。
- “=”后的目标变量名是引用的目标，声明引用时，必须同时对其进行初始化，即为其赋值，否则会产生编译错误。

例如，下列语句声明了一个对整型变量 a 的引用 ra。

```
int a;
int &ra=a;                                     //定义引用 ra,它是变量 a 的引用，即别名
```



在进行引用声明时，需要注意如下几个事项：

- 引用不是值。不占用内存空间，声明引用时，目标的存储状态不会改变。所以，引用只有声明，没有定义。
- 不能建立数组的引用。因为数组是一个由若干个元素所组成的集合，所以无法建立一个数组的别名。



注意 引用声明完毕后，相当于目标变量名有两个名称，即该目标原名称和引用名，不能再把该引用名作为其他变量名的别名。

17.1.2 引用的使用

17.1.1 节介绍了引用的声明方式，本节将讲解其使用。事实上，声明对某一个变量的引用后，就可以像变量一样使用了，而且其值与变量的值同步变化。

【范例 17-1】引用的使用。该范例声明了一个变量 `a` 和该变量的引用 `ra`，并为该变量 `a` 赋值，实现代码如代码清单 17-1 所示。

代码清单 17-1

```
1  #include <iostream.h>
2  void main()
3  {
4      int a;                                //声明变量
5      int &ra=a;                            //声明引用并赋初值
6      a=1;                                  //给变量赋值
7      cout<<"给变量a 赋值 1 后: "<<endl;
8      cout<<"a= " <<a<<endl;              //输出
9      cout<<"ra= " <<ra<<endl;
10     ra=2;                                //给引用赋值
11     cout<<"给引用 ra 赋值 2 后: "<<endl;
12     cout<<"ra= " <<ra<<endl;
13     cout<<"a= " <<a<<endl;
14 }
```

【运行结果】在 Visual C++中指向上述程序，其返回结果如图 17-2 所示。

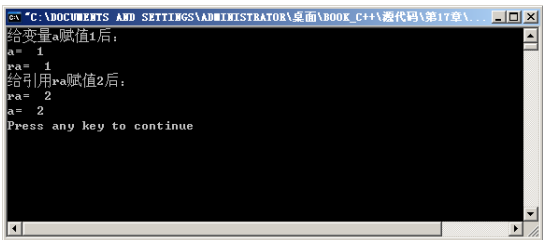


图 17-2 声明引用

【范例解析】上述程序中，声明了 `ra` 是对整型变量 `a` 的引用，在对变量 `a` 进行赋值 1 后，引用 `ra` 没有赋值，但由于 `ra` 是 `a` 的别名，因此 `ra` 的值也为 1。对引用 `ra` 赋值 2 后，再查看变量 `a` 的值，发现其也变为 2 了。读者可以看出，上述程序中的赋值语句：

`a=1;`

事实上就等价于

`ra=1;`



声明一个引用，不是新定义了一个变量，它只表示该引用名是目标变量名的一个别名，它本身不是一种数据类型，因此引用本身不占存储单元，系统也不给引用分配存储单元。因此，对引用求地址，就是对目标变量求地址，&ra 与 &a 相等。

17.2 引用的操作

通过 17.1 节内容的学习，读者知道了引用只是某个变量或目标的同义词，那它有什么用途呢？本节将讨论引用的两个主要用途：作为函数的参数及从函数中的返回值。

17.2.1 引用作为函数参数

引用在 C++ 中引入，其中的一个重要作用就是作为函数的参数。以前的 C 语言中函数参数传递是值传递，如果有大块数据作为参数传递的时候，采用的方案往往是指针，因为这样可以避免将整块数据全部压栈，可以提高程序的效率。但是 C++ 中又增加了一种同样有效率的选择，甚至在某些特殊情况下是必需的选择，这就是引用。

第 5 章函数中就简要介绍过函数参数的引用传递，此处同样通过交换两个整型数值的函数 swap 来理解引用作为函数参数的实现。

【范例 17-2】引用作为参数。该范例定义了函数 swap()，其定义的形式参数 p1 和 p2 就是引用，而在 main() 函数中调用 swap() 函数，实现代码如代码清单 17-2 所示。

代码清单 17-2

```

1  #include <iostream.h>
2  void swap(int &p1, int &p2)           //此处函数的形参 p1、p2 都是引用
3  {
4      int p;                           //定义整型变量 p
5      p=p1;                             //交换两个变量的值
6      p1=p2;
7      p2=p;
8  }
9  void main()
10 {
11     int a,b;
12     cout<<"Please input 2 number: "<<endl;
13     cin>>a>>b;                         //输入 a、b 两变量的值
14     cout<<"Before swap:"<<endl;
15     cout<<"a= "<<a<<endl;              //输出交换前的值
16     cout<<"b= "<<b<<endl;
17     swap(a,b);                         //直接以变量 a 和 b 作为实参调用 swap 函数
18     cout<<"After swap:"<<endl;
19     cout<<"a= "<<a<<endl;              //输出结果
20     cout<<"b= "<<b<<endl;
21 }
```

【运行结果】在 Visual C++ 中指向上述程序，其运行结果如图 17-3 所示。

【范例解析】读者可以看出，在程序中调用引用传递参数的函数时，在相应的主调函数调用点处，直接以变量作为实参进行调用即可，而不需要对实参变量有任何的特殊要求。

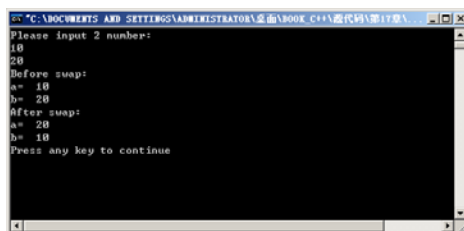


图 17-3 引用作为参数



17.2.2 引用作为返回值

事实上，引用除了 17.2.1 节介绍的作为函数的参数外，还可以使用引用返回函数值。引用作为函数的返回值对函数有一定的要求，它要求函数定义时要按以下格式来定义：

```
类型标识符 &函数名(形参列表及类型说明)
{
    函数体
}
```

其中，类型标识符为函数返回值的类型，可以是前面介绍的各种基本数据类型；函数名为符合 C++ 标识符命名规则的所有标识符。



以引用返回函数值，定义函数时需要在函数名前加“&”符号。用引用返回一个函数值的最大好处是在内存中不产生被返回值的副本。

【范例 17-3】引用作为函数返回值。该范例定义了一个普通的函数 func1，它用返回值的方法返回函数值，另外一个函数 func2，以引用的方法返回函数值，实现代码如代码清单 17-3 所示。

代码清单 17-3

```
1  #include <iostream.h>
2  float temp; //定义全局变量 temp
3  float func1(float r); //声明函数 func1
4  float &func2(float r); //声明函数 func2
5  float func1(float r) //定义函数 func1，它用返回值的方法返回函数值
6  {
7      temp=(float)(r*r*3.14); //强制类型转换
8      return temp;
9  }
10 float &func2(float r) //定义函数 func2，它用引用方式返回函数值
11 {
12     temp=(float)(r*r*3.14);
13     return temp;
14 }
15 void main() //主函数
16 {
17     float a=func1(5.0); //第 1 种情况，系统生成要返回值的副本（临时变量）
18     float b=func2(5.0); //第 2 种情况，系统不生成返回值的副本
19                         //可以从被调函数中返回一个全局变量的引用
20     float &c=func2(5.0); //第 3 种情况，系统不生成返回值的副本
21                         //可以从被调函数中返回一个全局变量的引用
22     cout<<"a= "<<a<<endl;
23     cout<<"b= "<<b<<endl;
24     cout<<"c= "<<c<<endl;
25 }
```

【运行结果】在 Visual C++ 中运行上述程序，其执行结果如图 17-4 所示。

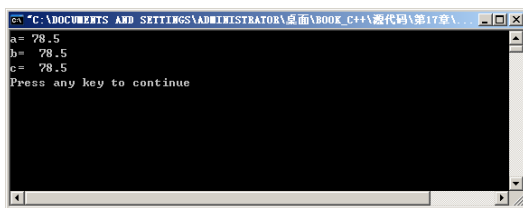


图 17-4 引用作为返回值

【范例解析】上述程序中，func2 函数使用了引用作为返回值，而在主函数 main()中就定义了一个引用 c 用于接收其返回值。一般来说，以引用作为返回值，必须遵守以下规则：

- 不能返回局部变量的引用。
- 不能返回函数内部 new 分配的内存的引用。例如，被函数返回的引用只是作为一个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（即由 new 分配的空间）就无法释放，会造成内存溢出等错误。
- 可以返回类成员的引用。

17.3 动态内存分配

动态内存分配是指在程序的运行期间可以根据实际需要随时申请内存和释放内存。实行动态内存管理可以节省内存空间，提高程序运行效率。在 C++ 中，提供了两种运算符进行动态内存管理：new 和 delete 运算符。将 new 运算符与 delete 运算符一起使用，就可以直接进行动态内存的申请和释放。

17.3.1 申请动态内存

在 C++ 中，new 运算符用于申请所需的内存单元，返回指定类型的一个指针，在有些参考资料中也称为创建内存单元。一般来说，new 运算符的语法格式为：

指针=new 数据类型；

其中，指针应预先声明，指针指向的数据类型与 new 后的数据类型相同。若申请成功，则返回分配单元的首地址给指针；否则（比如没有足够的内存空间）返回 0（一个空指针）。例如，下列语句申请一个整型数据空间：

```
int *p;
p=new int;
```

执行上述语句后，系统将自动根据 int 类型的空间大小开辟一个内存单元，用来保存 int 型数据，并将地址保存在指针 p 中。



实际程序中，可以用 new 运算符申请一块保存数组的内存单元，即创建一个数组。创建一维数组的语法格式为：

指针=new 数据类型[常量表达式]；

其中，常量表达式给出数组元素的个数，指针指向分配的内存首地址，指针的类型与 new 后的数据类型相同。例如，下面语句申请一个包含 10 个元素的整型数组空间。

```
int *p;
p=new int[10];
```

执行上述语句后，系统为指针 p 分配了整型数组的内存，数组中有 10 个元素。对于动态创建多维数组，情况要复杂一些。以二维数组为例，语法格式为：

指针=new 数据类型[常量表达式 1][常量表达式 2]；

若申请成功，指针指向分配的内存首地址，但此时指针的类型不是 new 后的数据，而是一个该类型的数组，即指针是一个数组指针。例如，

```
int (*p)[3];
p=new int[2][3];
```

执行上述语句后，系统为指针 p 分配了一个二维数组。



【范例 17-4】new 运算符的使用。该范例定义了三个指针，其分别申请了一个整型空间、一个包含 5 个元素的整型数组空间和一个包含 6 个元素的整型二维数组空间，并输出其中的一个元素，实现代码如代码清单 17-4 所示。

代码清单 17-4

```
1  #include <iostream.h>
2  void main()
3  {
4      int *pa;                                //定义整型指针
5      pa=new int;                             //申请一个空间
6      *pa = 100;                             //指针初始化
7      cout<<"*pa= " <<*pa <<endl;
8      int *pb;
9      pb=new int[5];                         //申请一个数组空间
10     for(int i=1;i<5;i++)
11     {
12         *(pb+i)=i;                         //指针初始化
13     }
14     cout<<"*(pb+2)= " <<*(pb+2)<<endl;    //输出
15     int (*pc)[3];
16     int j;
17     pc=new int[2][3];                     //申请一个二维数组空间
18     for (i=0;i<2;i++)
19         for(j=0;j<3;j++)
20             *(pc[i]+j)=1;                 //指针初始化
21     cout<<"*(pc[1]+1)= " <<*(pc[1]+1)<<endl;
22 }
```

【运行结果】在 Visual C++ 中运行上述程序，其运行结果如图 17-5 所示。

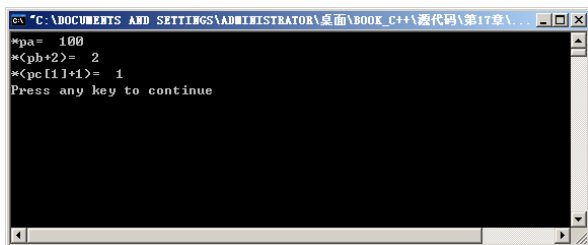


图 17-5 new 运算符

【范例解析】上述程序中，指针 pa 指向新申请的一个整型空间，为其赋值 100 后输出结果为 100；指针 pb 指向新申请的一个数组空间，其包含 5 个整型元素，为其赋值 1~5，输出第二个元素结果为 2；指针 pc 指向新申请的一个二维数组空间，其包含 6 个整型元素，均对其初始化为 1，输出其中的 pc[1][1] 元素为 1。



注意 用 new 运算符申请空间非常方便。事实上，new 运算符在实际程序中应用很频繁，读者应仔细理解。

17.3.2 释放空间

与 new 运算符相反，delete 运算符是释放 new 申请到的内存。即当程序中不再需要使用运算符 new 创建的某个内存单元时，就必须用运算符 delete 来删除它，其语法格式为：

```
delete 指针;           //释放非数组内存单元
```

```
delete[常量] 指针; //释放数组内存单元
```

其中, 指针是指向需要释放的内存单元的指针的名字。并且 `delete` 只是删除动态内存单元, 并不会将指针本身删除。

释放对象的不同, `delete` 的语法格式也不同。数组内存单元的释放一定要带[常量]部分, 常量告诉 `delete` 数组有多少个元素。如果没有带[常量]部分, 则只释放数组的第一个元素占据的内存单元。

- 对 `int` 型内存单元的申请和释放, 代码如下:

```
int *p;
p=new int; //申请内存单元
*p=1;
delete p; //释放内存单元
```

- 对数组内存单元的申请和释放, 代码如下:

```
int *p;
p=new int[10];
delete[10] p;
```

如上对数组内存单元的释放中就添加了常量 10, 表示释放指针 `p` 所指向数组的 10 个空间, 如没有加该常量, 则系统只释放该数组的第一个元素所占据的空间。使用 `new` 运算符和 `delete` 运算符进行内存空间的动态分配是很方便的, 但是, 在实际的程序中, 读者需要注意以下三个事项:

- 在程序中对对应于每次使用运算符 `new`, 都应该相应地使用运算符 `delete` 来释放申请的内存。并且对应于每个运算符 `new`, 只能调用一次 `delete` 来释放内存, 否则有可能导致系统崩溃。
- 运算符 `delete` 必须用于先前 `new` 分配的有效指针, 而不能用于未定义的其他任何类型的指针。
- 对空指针调用 `delete` 是安全的。

【范例 17-5】`delete` 运算符的使用。该范例实现用 `new` 运算符所申请空间的释放, 包括释放一个内存单元和多个内存单元的实现, 代码如代码清单 17-5 所示。

代码清单 17-5

```
1  #include <iostream.h>
2  void main()
3  {
4      int *pa; //定义整型指针
5      pa=new int; //申请一个内存单元
6      cout<<"已申请一个内存单元"<<endl;
7      *pa=1; //赋值
8      delete pa; //释放一个内存单元
9      cout<<"已删除一个内存单元"<<endl;
10     int *pb;
11     pb=new int[10]; //申请一个内存单元
12     cout<<"申请一个包含 10 个元素的数组单元"<<endl;
13     delete[10] pb;
14     cout<<"已删除该数组内存单元"<<endl; //释放该数组内存单元
15 }
```

【运行结果】在 Visual C++ 中运行上述代码, 其结果如图 17-6 所示。

【范例解析】上述程序中, 分别用 `new` 运算符申请了一个整型内存空间和一个包含 10 个元素的整型数组空间, 再用 `delete` 运算符分别将其释放。

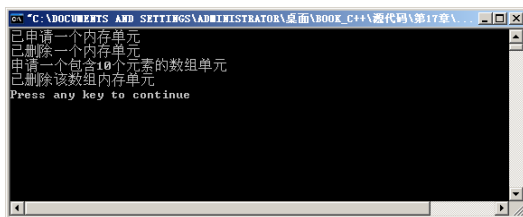


图 17-6 delete 运算符



注意 C++语言保留了 C 语言中的两个库函数：`malloc()` 与 `free()`。这两个函数也是实现动态内存分配作用的，其功能分别与运算符 `new` 和 `delete` 相似，下面章节将讨论这两个运算符。但是最好不要将库函数和运算符混合使用，否则可能会导致系统崩溃。

17.3.3 malloc 和 free 库函数

有过 C 语言学习经验的读者应该知道，`malloc` 和 `free` 是 C 语言的标准库函数。前面介绍的 `new` 和 `delete` 是 C++ 的运算符。这两种形式都可用于申请动态内存和释放内存。既然库函数 `malloc()` 和 `free()` 可以实现动态内存的管理，那为什么还要 `new` 和 `delete` 运算符呢？

这是因为对于非内部数据类型的对象而言，只用库函数 `malloc()` 和 `free()` 无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于 `malloc()` 和 `free()` 是库函数而不是运算符，它们不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于 `malloc` 和 `free` 函数。

因此 C++ 语言需要一个能完成动态内存分配和初始化工作的运算符 `new`，以及一个能完成清理与释放内存工作的运算符 `delete`。为了使读者更好地理解动态内存管理，这里先看看使用库函数 `malloc/free` 和运算符 `new/delete` 分别是如何实现对象的动态内存管理的。

【范例 17-6】 使用库函数和运算符实现动态内存管理。该范例使用 `malloc` 和 `free` 库函数实现动态内存管理，实现代码如代码清单 17-6 所示。

代码清单 17-6

```

1  #include <iostream>
2  using namespace std;                      //using 指令指明 std 命名空间
3  class Obj                                  //定义类
4  {
5  public :
6      Obj(void)                             //定义构造函数
7      {
8          cout << "Initialization" << endl;
9      }
10     ~Obj(void)                             //定义析构函数
11     {
12         cout << "Destroy" << endl;
13     }
14     void Initialize(void)                   //定义成员函数
15     {
16         cout << "Initialization" << endl;
17     }
18     void Destroy(void)                     //定义成员函数
19     {
20         cout << "Destroy"<< endl;
21     }

```

```

22  };
23  void UseMallocFree(void)                //定义使用库函数的函数
24  {
25      Obj *a = (Obj *)malloc(sizeof(Obj)); //申请动态内存
26      a->Initialize();                     //初始化
27      //...
28      a->Destroy();                         //清除工作
29      free(a);                             //释放内存
30  }
31  void UseNewDelete(void)                 //定义使用运算符的函数
32  {
33      Obj *a = new Obj;                   //申请动态内存并且初始化
34      //...
35      delete a;                           //清除并且释放内存
36  }
37  void main()
38  {
39      Obj obj;                             //创建对象, 调用构造函数
40      UseMallocFree();                     //调用函数
41      UseNewDelete();
42  }

```

【运行结果】在 Visual C++ 中指向上述程序，其运行结果如图 17-7 所示。

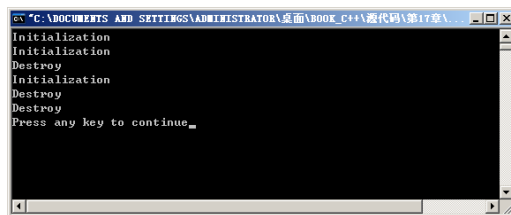


图 17-7 使用库函数和运算符实现动态内存管理

【范例解析】上述代码中，类 Obj 的函数 Initialize 模拟了构造函数的功能，函数 Destroy 模拟了析构函数的功能。函数 UseMallocFree 中，由于 malloc/free 不能执行构造函数与析构函数，必须调用成员函数 Initialize 和 Destroy 来完成初始化与清除工作。而相比之下，函数 UseNewDelete 实现初始化和释放内存则比其简单得多。

因此，在实际的程序中，应尽量少用 malloc/free 来完成动态对象的内存管理，而使用 new/delete。如果用 free 释放“new 创建的动态对象”，那么该对象因无法执行析构函数而可能导致程序出错。如果用 delete 释放“malloc 申请的动态内存”，理论上讲程序不会出错，但是该程序的可读性很差。所以 new/delete 必须配对使用，malloc/free 也一样。



注意 new/delete 不是 C++ 的库函数，而是运算符，而 malloc/free 才是 C 和 C++ 的标准库函数。

17.4 const 引用

在前面关于数据类型的介绍中，给读者介绍了 const 修饰符，它用于定义常量。事实上，引用也可以在前面加上 const 修饰符，称为 const 引用，用以区别普通的 C++ 引用。

这里将 const 引用单独列出是由于与普通的 C++ 引用不同。简单地说，const 引用表示指向 const 对象的引用，而非 const 引用表示指向非 const 对象的引用。例如，下面语句定义了常量 ival 和 const 引用：



```
const int ival = 1024;           //定义 const 常量 ival
const int &ref1 = ival;          //引用和对象都是 const
```

如果定义了一个非 const 引用，而将 const 对象赋给该引用，这是非法的。因为对引用的修改将改变 const 对象的值，而 const 对象是不能改变值的。例如，针对上述语句，添加下面的语句是错误的：

```
int &ref2 = ival;                //是错误的，因为对 ref2 修改会导致修改 ival 的值
```



一般来说，非 const 引用只能绑定到与该引用同类型的对象；const 引用则可以绑定到不同但相关的类型（该类型能够转换到引用的类型）的对象，甚至绑定到右值。

【范例 17-7】const 引用。该范例定义了 const 引用，其可以绑定不同类型的对象，读者可观察其使用，实现代码如代码清单 17-7 所示。

代码清单 17-7

```
1  #include <iostream>
2  using namespace std;           //使用命名空间
3  void main()
4  {
5      int var = 6;                //定义整型变量并初始化
6      cout<<"整型变量 var= "<<var<<endl;
7      //int& r = 9;              //引用的错误
8      const int& r1 = 9;          //声明 const 引用，正确
9      cout<<"const 引用 r1= "<<r1<<endl;    //输出
10     //int &r2=r1;              //错误
11     const int &r2=r1;          //正确
12     cout<<"const 引用 r2= "<<r2<<endl;
13     const int& r3 = var + r1;    //正确
14     cout<<"const 引用 r3= "<<r3<<endl;    //输出引用的结果
15 }
```

【运行结果】在 Visual C++ 中指向上述程序，其运行结果如图 17-8 所示。

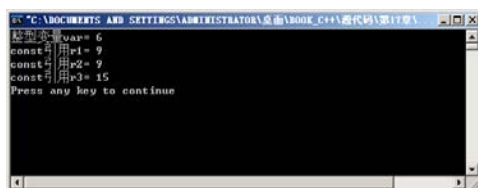


图 17-8 const 引用

【范例解析】上述代码中，定义了整型变量 var 和 const 引用 r1、r2 和 r3，读者可以看出，只有 const 引用才能直接为其赋值 const 常量。此外，const 引用还可用于普通变量与 const 引用进行运算，如上述代码中的第 13 行。

如果在上述代码中将第 7 行和第 10 行的注释符去掉，即给非 const 引用赋值常量或 const 引用，其将出现如图 17-9 所示的编译错误。

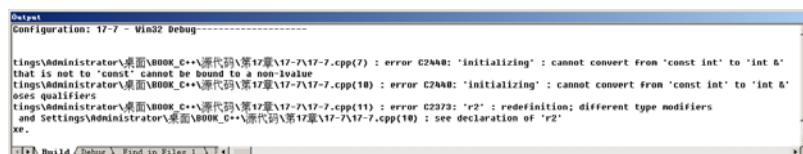


图 17-9 编译错误

`const` 引用则可以绑定到不同但相关类型的对象，甚至绑定到右值。这是因为如果是绑定到不同类型的对象或右值，会先生成一个与引用类型相同的临时对象（必要时进行类型转换），然后再引用变量与该临时对象绑定。由于临时对象是右值，所以只能进行 `const` 引用。另外，还可以从另一个角度来理解，例如：

```
double var = 2.718;
const int& r = var;
```

上面的代码等价于：

```
double var = 2.718;
int tmp = var;
const int& r = tmp;
```

如果 `r` 不是 `const` 引用，那么可以修改 `r` 的值。这样做，事实上是修改了临时变量 `tmp` 的值，而这与程序员试图修改 `var` 的值的意愿完全偏离。这种错误是很难察觉的。而因为 `const` 引用不能修改，所以完全避免了这种错误的发生。

17.5 指针与引用的区别

在第 8 章时学习过指针，读者应对指针有一定的了解，那么指针和引用有什么区别和联系呢？指针其实就是一个变量，和其他类型的变量是一样的，它是一个占用四字节的变量（32 位机上），与其他变量的不同之处就在于它的变量值是一个内存地址，指向内存的另外一个地方。而引用则是变量的一个别名。



一个指针变量可以指向 `NULL`，表示它不指向任何变量地址，但是引用必须在声明的时候就就得和一个已经存在的变量相绑定，而且这种绑定不可改变。

【范例 17-8】指针与引用的区别。该范例指出了引用和指针在定义及使用上的区别，实现代码如代码清单 17-8 所示。

代码清单 17-8

```
1  #include <iostream>
2  using namespace std;                //使用命名空间
3  void main(int argc, char* argv[])
4  {
5      int ival = 1024;
6      int *pi= &ival;                //定义指针
7      int &rval = ival;               //定义引用
8      int jval = 4096;
9      int xval = 8192;
10     cout << "ival = " << ival << "\t";    //输出变量值
11     cout << "&ival = " << &ival << "\t";    //输出地址
12     cout << endl;
13     cout << "pi = " << pi << "\t";        //输出指针变量
14     cout << "&pi = " << &pi << "\t";        //输出指针地址
15     cout << "**pi = " << *pi << "\t";        //输出指针值
16     cout << endl;
17     cout << "rval = " << rval << "\t";        //输出引用
18     cout << "&rval = " << &rval << "\t";    //输出引用值
19     cout << endl;
20     cout << "jval = " << jval << "\t";        //输出变量值
21     cout << "&jval = " << &jval << "\t";    //输出变量地址
22     cout << endl;
```




23 }

【运行结果】在 Visual C++ 中指向上述程序，其运行结果如图 17-10 所示。

```

C:\DOCUMENTS AND SETTINGS\ADMINISTRATOR\桌面\BOOK C++\源代码\第17章\...
ival = 1024      &ival = 0012FF7C
pi = 0012FF7C   &pi = 0012FF78 *pi = 1024
rval = 1024     &rval = 0012FF7C
jval = 4096     &jval = 0012FF70
Press any key to continue
  
```

图 17-10 指针与引用的区别

【范例解析】上述代码中，定义了指针 `pi` 和引用 `rval`，`pi` 是一个指针变量，它指向整型变量 `ival`，但 `pi` 本身的值为一个十六进制的地址值，而 `&pi` 为存储该地址值的地址，`*pi` 则为整型变量 `ival` 的值。`rval` 为引用，其是整型变量 `ival` 的一个别名，因此 `&rval` 的值即为存储变量 `ival` 的值，都是指存储 `ival` 变量的地址。

17.6 小结

本章主要讨论了有关 C++ 中引用和内存管理的相关内容。引用是 C++ 引入的新语言特性，是 C++ 常用的一个重要内容之一。本章主要就引用的概念和使用，以及引用在 C++ 程序中的作用做了详细讲解。引用主要用于传递函数的参数和给出返回值。此外，动态内存管理也是 C++ 的一个重要特点，其提供了标准库函数和运算符进行动态内存的分配和释放，本章将两者进行了对比，使得读者了解 `new/delete` 运算符更适合进行内存管理，最后通过一个较为详细的实例介绍了使用 `new/delete` 运算符如何动态地进行内存分配和释放。

17.7 习题

1. 引用在 C++ 程序中用做参数函数和返回值，分别是如何实现的？

【解答】在程序中调用引用传递参数的函数时，在相应的主调函数调用点处，直接以变量作为实参进行调用即可，而不需要对实参变量有任何的特殊要求。以引用返回函数值，定义函数时需要在函数名前加“&”符号。用引用返回一个函数值的最大好处是在内存中不产生被返回值的副本。

2. 下列程序的执行结果是什么？

```

#include<iostream.h>
void main(.
{
    int n=10;
    int* pn=&n;
    int &rn=n;
    *pn++;
    cout<<"n="<<n<<endl;
    rn++;
    cout<<"n="<<n<<endl;
}
  
```

【解答】该习题主要考查引用和指针的具体应用。上述程序中，定义了指针 `pn` 和引用 `rn`，其中指针 `pn` 执行变量 `n`，引用 `rn` 是 `n` 的别名，分别对其进行++运算后，将 `n` 的值输出。由于指针变化后，其指向的值 `n` 并不会随着变化，而引用变化了，其被引用值随之变化，因此输出结果为：`n=10 n=11`。

3. 下面程序的运行结果是什么?

```
#include<iostream.h>
void fun(int &a,int &b)
{
    int p;
    p=a;
    a=b;
    b=p;
}
void exchange(int &a,int &b,int &c)
{
    if(a<b) fun(a,b);
    if(a<c) fun(a,c);
    if(b<c) fun(b,c);
}
int main()
{
    int a,b,c;
    a=12;
    b=639;
    c=78;
    exchange(a,b,c);
    cout<<"a="<<a<<" ,b="<<b<<" ,c="<<c<<endl;
}
```

【解答】该习题主要考查引用作为函数参数传递的应用。上述程序中定义了两个函数 `fun` 和 `exchange`，都分别使用了引用作为函数的参数，在调用时实参不需做任何变化。根据对函数的分析，读者可以看到其功能是比较参数的大小，该程序段输出从大到小的 3 个变量的值。因此，程序输出结果为：`a=639,b=78,c=12`。

4. 编写一个 C++ 程序，在包含类的情况下调用 `new` 和 `delete` 运算符实现内存的动态分配，并判断分配是否成功。

【解答】该习题主要考查 `new` 和 `delete` 运算符的应用。此外，程序必须首先定义一个类，并包含构造函数，在主函数定义对象后动态分配内存空间，并通过 `if` 语句判断分配是否成功。其简要的实现代码如下所示。

<code>#include <iostream></code>	<code>//使用命名空间</code>
<code>using namespace std;</code>	<code>//定义 CPoint 类</code>
<code>class CPoint</code>	
<code>{</code>	
<code>public:</code>	<code>//定义公有成员</code>
<code> CPoint(int x=0,int y=0);</code>	<code>//构造函数，默认参数 x=0,y=0</code>
<code> int SetTemp();</code>	<code>//设置 m_temp 的值</code>
<code> void Print();</code>	<code>//显示，一般成员函数</code>
<code> void Print() const;</code>	<code>//显示，常成员函数</code>
<code>private:</code>	<code>//定义私有成员</code>
<code> int m_x;</code>	<code>//定义整型变量成员</code>
<code> const int m_y;</code>	<code>//定义静态成员</code>
<code> static const int m_z;</code>	
<code> int m_temp;</code>	<code>//增加的数据成员，没有初始化</code>
<code>};</code>	
<code>void main()</code>	<code>//测试主函数</code>
<code>{</code>	
<code> const int ARRAY_SIZE=5;</code>	<code>//定义静态变量</code>
<code> CPoint *pCPoint;</code>	<code>//创建对象指针</code>
<code> pCPoint=new CPoint[ARRAY_SIZE];</code>	<code>//创建 5 个对象，创建时会自动调用构造函数</code>
<code> if(pCPoint==NULL)</code>	<code>//申请空间失败</code>



```
{
    cout<<"动态内存分配失败。\\n";           //错误提示
    exit(0);                                   //退出程序
}
for(int iCnt=0;iCnt<ARRAY_SIZE;iCnt++) //进入循环
{
    pCPoint[2].SetTemp();                     //只修改对象 2 的 m_temp 值
    pCPoint[iCnt].Print();                     //输出结果
}
delete[] pCPoint;                             //释放对象数组
//动态释放后，让指针指向 NULL，否则，它依然指向原动态空间
pCPoint=NULL;                                 //指针执行空
CTemp *pCTemp=new CTemp(5);                   //申请 5 个单元地址空间
if(pCTemp==NULL)                               //申请空间失败
{
    cout<<"动态内存分配失败。\\n";           //错误提示
    exit(0);                                   //退出程序
}
//pCTemp=new CTemp[ARRAY_SIZE];               //错误，CTemp 没有默认参数构造函数
delete pCTemp;                                //释放单个对象
}
```

第五篇 C++编程实践篇

第 18 章 标准模板库 STL

STL (Standard Template Library), 即标准模板库, 是一个具有工业强度的、高效的 C++ 程序库。它被容纳于 C++ 标准程序库 (C++ Standard Library) 中, 是 ANSI/ISO C++ 标准中最新的也是极具革命性的一部分。该库包含了诸多在计算机科学领域里所常用的基本数据结构和基本算法, 为广大 C++ 程序员们提供了一个可扩展的应用框架, 高度体现了软件的可复用性。这种现象有些类似于 Microsoft Visual C++ 中的 MFC (Microsoft Foundation Class Library), 或者是 Borland C++ Builder 中的 VCL (Visual Component Library)。本章将主要介绍 STL 的相关概念及其组成, 使读者对 STL 有大致了解。

以下是对读者在学习本章内容时所提出的几个基本要求, 也是本章希望能够达到的目的, 让读者在学习本章内容时可以作为一个学习的参照。

- 了解标准模板库 STL 的基本概念及其在 C++ 程序设计中的作用。
- 掌握常用的 STL 容器的类别及其相关应用。
- 掌握算法和迭代器的使用。

18.1 标准模板库

STL 是最新的 C++ 标准函数库中的一个子集, 这个庞大的子集占据了整个库大约 80% 的分量。而作为在实现 STL 过程中扮演关键角色的模板则几乎充斥了整个 C++ 标准函数库。此处有必要看一看 C++ 标准函数库里包含了哪些内容, 其中又有哪些是属于标准模板库 (STL) 的。

18.1.1 STL 概述

STL (Standard Template Library, 标准模板库) 是惠普实验室开发的一系列软件的统称。它是由 Alexander Stepanov、Meng Lee 和 David R Musser 在惠普实验室所开发出来的。现在虽说它主要出现在 C++ 中, 但在被引入 C++ 之前该技术就已经存在了很长的一段时间。

STL 提供了一系列具有良好结构的通用 C++ 组件, 这些组件提供了强大的功能。标准库的设计必须确保所有的模板算法既能操作库中的数据类型, 也能操作 C++ 固有的数据类型。例如, 所有的算法都适用于普通指针类型。库中各组件功能是独立的, 或者说, 用户可以自己设计算法操作库提供的数据结构, 也可以使用标准库的算法操作自定义的数据类型。

简单地说, STL 在 C++ 程序设计中的作用是提供一个可供函数调用的组件和函数库, 当需要时通过接口来调用, 如图 18-1 所示。



图 18-1 STL 的作用



提示 STL 的目的是标准化组件, 用户不用重新开发它们, 而可以仅仅使用这些现成的组件。STL 现在是 C++ 的一部分, 因此不用额外安装什么, 其被内建在编译器之内。

18.1.2 STL 的引入

前面讲到了 STL 的许多特点和优势, 现在来看一下 STL 在实际程序中起到什么作用, 本节将通过一个代码段的比较来引入 STL, 体现其优势。



【范例 18-1】第一个 STL 程序。该范例中调用了 STL 中的函数，实现代码如代码清单 18-1 所示。

代码清单 18-1

```
1  #include <vector>                                //包含头文件
2  #include <iostream>
3  int main()
4  {
5      std::vector<double> a;                        //定义数据变量
6      std::vector<double>::const_iterator i;        //定义变量
7      a.push_back(1);                               //调用 STL 中函数
8      a.push_back(2);
9      a.push_back(3);
10     a.push_back(4);
11     a.push_back(5);                               //调用 STL 中函数
12     for(i=a.begin(); i!=a.end(); ++i)             //循环输出
13     {
14         std::cout<<(*i)<<std::endl;               //输出该容器内所有元素
15     }
16     return 0;
17 }
```

【运行结果】在 Visual C++ 中新建一个【C++ Source file】，输入如上的代码，编译运行无误后，其结果如图 18-2 所示。

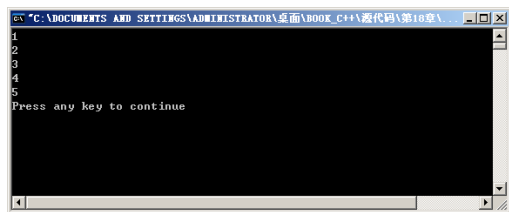


图 18-2 第一个 STL 程序

【范例解析】上述代码中，第 1 行预编译了一个以前未提到过的头文件，这个头文件在以后会详细介绍，在代码的第 5~6 行使用了一对<>符号，它表示模板，它的作用在第 19 章中将详细介绍。读者通过上述运行结果可以看出，其输出 5 个数字。

读者可以看到，上述代码中第 7~11 行使用了 push_back() 函数，第 12 行使用了 begin() 和 end() 函数，而这些函数在程序中并没有被定义过，却可以使用，这是因为这些函数已经被头文件 vector.h 所包含，如同 strcpy() 函数被 string.h 头文件所包含一样。



提示 这是第一个使用了 STL 的 C++ 程序，读者可以看出，该程序实现了将 5 个常数输入进向量 a，并在最后将其输出。

18.1.3 STL 的组成

为了理解 C++ 中如何使用 STL，读者需要先了解 STL 的组成。读者知道，虽然 STL 是一个模板库，但其中也包含了许多部分。一般来说，STL 由如下的 6 大部分组成。

- 容器 (Containers): 用于管理数据集合，其包括各种数据结构，比如 vector、list、deque、set、map 用来存放数据，从实现的角度来看，STL 容器是一种 class template。
- 算法 (Algorithms): 定义了计算过程，其包括各种算法，比如 sort、search、copy、erase 等。从实现的角度来看，STL 算法是一种 function template。
- 迭代器 (Iterators): 提供遍历容器的方法，它扮演了容器与算法之间的胶合剂，是所谓的“泛型指针”。共有 5 种类型及其他衍生变化。从实现角度来讲 STL 迭代器是一

种将 `operator*`、`operator->`、`operator++`、`operator--` 等指针相关操作予以重载的 `class template`。所有的 STL 容器都附有自己专属的迭代器。

- 仿函数 (Functors): 将函数封装在对象中, 供其他组件使用。行为类似函数, 可作为算法的某种策略, 从实现角度来看, 它是一种重载了 `operator()` 的 `class` 或者 `class template`, 一般的函数指针可视为狭义的仿函数。
- 适配器 (Adapters): 一种用来修饰容器、仿函数或迭代器的接口, 如 STL 提供的 `queue` 和 `stack`, 虽然看似容器, 其实只能算是一种容器适配器, 因为它们底部完全借助 `deque`, 所有操作都由底层的 `deque` 供应。改变 functor 接口者称为 `function adapter`, 改变 container 接口者称为 `container adapter`。
- 配置器 (Allocators): 负责空间配置与管理, 从实现角度来看配置器是一个实现了动态空间配置、空间管理、空间释放的 `class template`。

STL 的代码从广义上讲分为三类: `algorithm` (算法)、`container` (容器) 和 `iterator` (迭代器)。几乎所有的代码都采用了模板类和模板函数的方式, 这相比于传统的由函数和类组成的库来说提供了更好的代码重用机会。简单来说, 它们的关系如图 18-3 所示。

在 C++ 标准中, STL 被组织为下面的 13 个头文件: `<algorithm>`、`<deque>`、`<functional>`、`<iterator>`、`<vector>`、`<list>`、`<map>`、`<memory>`、`<numeric>`、`<queue>`、`<set>`、`<stack>` 和 `<utility>`。下面简单介绍一下 STL 各个部分的主要特点。

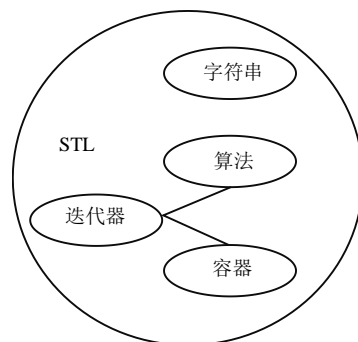


图 18-3 STL 的组成

18.2 算法

18.1 节中提到了, STL 主要包含算法、容器和迭代器。其中, STL 提供了大约 100 个实现算法的模板函数, 用户可以通过调用一两个算法模板就可以完成所需要的功能, 这样大大地提高了用户使用 C++ 进行程序设计的效率。

一般来说, STL 中的算法部分主要由头文件 `<algorithm>`、`<numeric>` 和 `<functional>` 组成。其中, 头文件 `<algorithm>` 由一大堆模板函数组成, 常用的函数涉及比较、交换、查找等。下面的实例就需要用到该头文件中的算法。



注意 头文件 `<numeric>` 体积很小, 只包括几个在序列上面进行简单数学运算的模板函数, 包括加法和乘法在序列上的一些操作。头文件 `<functional>` 中则定义了一些模板类, 用以声明函数对象。

【范例 18-2】 使用排序算法。范例使用 STL 算法中一个最为常用的算法——排序, 其实现代码如代码清单 18-2 所示。

代码清单 18-2

```
1  #include <iostream>
2  #include <algorithm>           //头文件 algorithm, 含有算法相关函数
3  #include <functional>
4  #include <vector>             //头文件 vector, 含有向量相关函数
5  using namespace std;         //使用命名空间
6  class myclass {               //定义类
7  public:
8      myclass(int a, int b):first(a), second(b){}
```



```

//构造函数
9     int first;
10    int second;
11    bool operator < (const myclass &m)const
//重载运算符<
12    {
13        return first < m.first;
14    }
15 };
16 bool less_second(const myclass &m1, const myclass &m2)
//根据第二个元素返回
17 {
18     return m1.second < m2.second;
19 }
20 int main()
//主函数
21 {
22     vector< myclass > vect;
//创建对象
23     for(int i = 0 ; i < 10 ; i ++ )
24     {
25         myclass my(10-i, i*3);
//创建对象，并初始化
26         vect.push_back(my);
//写入向量
27     }
28     for(i = 0 ; i < vect.size(); i ++ )
29     cout<<"("<<vect[i].first<<","<<vect[i].second<<")\n";
//输出未排序的向量
30     sort(vect.begin(), vect.end());
//调用排序算法
31     cout<<"after sorted by first:"<<endl;
32     for(i = 0 ; i < vect.size(); i ++ )
33     cout<<"("<<vect[i].first<<","<<vect[i].second<<")\n";
//输出按第一个值排序的结果
34     cout<<"after sorted by second:"<<endl;
35     sort(vect.begin(), vect.end(), less_second);
//调用排序算法
36     for( i = 0 ; i < vect.size(); i ++ )
37     cout<<"("<<vect[i].first<<","<<vect[i].second<<")\n";
//输出按第二个值排序的结果
38     return 0 ;
39 }

```

【运行结果】在 Visual C++ 中新建一个【C++ Source file】，输入如上的代码，编译运行无误后，其结果如图 18-4 所示。

```

C:\Documents and Settings\Administrator\桌面\BOOK C++\源代码\第18章\18-2\ch1
(10,0)
(9,3)
(8,6)
(7,9)
(6,12)
(5,15)
(4,18)
(3,21)
(2,24)
(1,27)
after sorted by first:
(1,27)
(2,24)
(3,21)
(4,18)
(5,15)
(6,12)
(7,9)
(8,6)
(9,3)
(10,0)
after sorted by second:
(10,0)
(9,3)
(8,6)
(7,9)
(6,12)
(5,15)
(4,18)
(3,21)
(2,24)
(1,27)
Press any key to continue_

```

图 18-4 使用排序算法

【范例解析】上述代码中，使用到了头文件<algorithm>和<functional>，对输出的向量分别按照其第一个元素值和第二个元素值进行升序排列。读者可以看到，上述代码并没有定义 sort() 函数，但在程序中两次调用了该函数，说明该函数已被头文件所包含。这两处调用的其参数个数是不同的，说明该函数被重载了。

提示 STL 中包含了大量的常用算法，其可供读者在需要的时候直接调用，只需知道该算法实现函数在哪个头文件下即可。

18.3 容器

在实际的开发过程中，数据结构本身的重要性不会亚于操作与数据结构的算法的重要性，当程序中存在对时间要求很高的部分时，数据结构的选择就显得更加重要。STL 容器允许重复利用已有的实现构造自己特定类型下的数据结构，通过设置一些模板类，这些模板的参数允许用户指定容器中元素的数据类型，从而可以提高编程效率。

18.3.1 什么是容器

容器部分主要由头文件<vector>、<list>、<deque>、<set>、<map>、<stack>和<queue>组成。对于常用的一些容器和容器适配器（可以看作由其他容器实现的容器），可以通过下表 18-1 总结一下它们和相应头文件的对应关系。

表 18-1 容器与头文件对应关系

数据结构	描 述	实现头文件
向量 (vector)	连续存储的元素	<vector>
列表 (list)	由节点组成的双向链表，每个节点包含着一个元素	<list>
双队列 (deque)	连续存储的指向不同元素的指针所组成的数组	<deque>
集合 (set)	由节点组成的红黑树，每个节点都包含着一个元素，节点之间以某种作用于元素对的谓词排列，没有两个不同的元素能够拥有相同的次序	<set>
多重集合 (multiset)	允许存在两个次序相等的元素的集合	<set>
栈 (stack)	后进先出的值的排列	<stack>
队列 (queue)	先进先出的值的排列	<queue>
优先队列 (priority_queue)	元素的次序是由作用于所存储的值对上的某种谓词决定的一种队列	<queue>
映射 (map)	由{键,值}对组成的集合，以某种作用于键对上的谓词排列	<map>
多重映射 (multimap)	允许键对有相等的次序的映射	<map>

读者可以看到，STL 容器中容纳了大量的数据结构模板，在前面使用的 vector 向量就是其中的一个模板，下面将详细介绍这些容器类。

18.3.2 向量

通过表 18-1 读者知道，向量是一种 vector 容器类，在前面的程序中就引用过该类。向量就像是盛放变长数组的花园，大约所有 STL 容器中有一半是基于向量的。通过前面的范例，读者知道，引入该向量类的方式如下：

```
#include <vector>
```




事实上, `vector` 是一种动态数组, 是基本数组的类模板。其内部定义了很多基本操作。既然这是一个类, 那么它就会有自己的构造函数。`vector` 类中定义了以下四种构造函数。

- 默认构造函数: 其构造了一个初始长度为 0 的空向量, 其调用如下:

```
vector<int> v1;
```

- 带有单个整型参数的构造函数: 此参数描述了向量的初始大小, 该构造函数还有一个可选的参数, 这是一个类型为 `T` 的实例, 描述了各个向量各种成员的初始值, 其调用如下:

```
vector<int> v2(init_size,0); //如果预先定义了 int init_size, 其成员值都被初始化为 0
```

- 复制构造函数: 构造一个新的向量, 作为已存在的向量的完全复制, 其调用如下:

```
vector<int> v3(v2);
```

- 带两个常量参数的构造函数: 产生初始值为一个区间的向量。区间由一个半开区间 `[first,last]` (MS word 的显示可能会有问题, `first` 前是一个左方括号, `last` 后面是一个右圆括号) 来指定, 其调用如下:

```
vector<int> v4 (first,last)
```

此外, 在实际程序中使用较多的还是向量类的成员函数, 其常用的成员函数如前面使用的 `begin()`、`end()` 等, 下面列出了其常用成员函数:

`begin()`、`end()`、`push_back()`、`insert()`、`assign()`、`front()`、`back()`、`erase()`、`empty()`、`at()`、`size()`

【范例 18-3】 使用向量。该范例是一个 `hello world` 程序, 其将一个字符串传送到一个字符向量中, 然后每次显示向量中的一个字符, 实现代码如代码清单 18-3 所示。

代码清单 18-3

```
1  // #include "stdafx.h"
2  #include <vector>                //STL 向量的头文件, 这里没有 "h"
3  #include <iostream>              //包含 cout 对象的头文件。
4  using namespace std; /          /保证在程序中使用 std 命名空间中的成员
5  char* szHW = "Hello World";    //这是一个字符数组, 以 "\0" 结束
6  int main(int argc, char* argv[])
7  {
8      vector <char> vec;           //声明一个字符向量 vector (STL 中的数组)
9      vector <char>::iterator vi; //为字符数组定义一个游标 iterator
10                                     //将一个指针指向 "Hello World" 字符串
11     char* cptr = szHW;
12     while (*cptr != '\0')
13     //初始化字符向量, 对整个字符串进行循环, 用来把数据填放到字符向量中, 直到遇到 "\0" 时结束
14     {
15         vec.push_back(*cptr); cptr++;
16         //push_back 函数将数据放在向量的尾部
17     }
18     for (vi=vec.begin(); vi!=vec.end(); vi++)
19         //将向量中的字符一个个地显示在控制台
20         // 这是 STL 循环的规范化的开始——通常是 "!=", 而不是 "<", 因为 "<" 在一些容器中没有定义
21         // begin() 返回向量起始元素的游标 (iterator), end() 返回向量末尾元素的游标 (iterator)
22         {
23             cout << *vi;          // 使用运算符 "*" 将数据从游标指针中提取出来
24         }
25     cout << endl;                // 换行
26     return 0;
```

【运行结果】在 Visual C++ 中新建一个【C++ Source file】，输入如上的代码，编译运行无误后，其结果如图 18-5 所示。

【范例解析】上述代码中，第 2 行代码调用了 STL 向量的头文件，第 9 行声明一个字符向量，并为该向量定义游标 iterator，它的功能类似于一个指针。代码第 11~15 行使用 while 循环语句依次取出将字符放在向量的尾部，第 18~23 行中使用 for 循环语句将其输出。

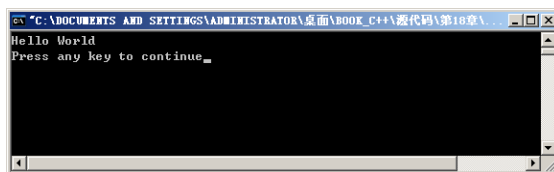


图 18-5 使用向量



注意 所谓游标其实是一个指针，用来指向 STL 容器中的元素，也能够指向其他的元素。

18.3.3 列表

列表也是容器类中的一种，其所控制的长度为 N 的序列是以一个有着 N 个节点的双向链表来存储的，支持双向迭代器，其预编译头文件如下：

```
#include <list>
```



提示 使用列表的优势是可以在链表中随意地插入和删除元素或是子链表，所需的仅是改变前后指针。但是，使用列表在定位操作中比如查找和随机存取，时间是线性增加的，而且在存储上每个节点还有前向和后向两个指针。

模板类 list 的另外一个特点是在异常处理方面，对任何容器来说，容器成员函数在执行中抛出的异常，使容器本身处于一种一致的状态，可以被销毁，并且容器没有对所分配的存储空间失去控制。但对大部分操作尤其是那些能够影响多个元素的操作，当抛出异常时，并没有制定容器的精确状态，但 list 保证对于大部分成员函数抛出异常时，容器能够恢复到操作前的状态。

列表类的定义如下：

```
typedef list<T, allocator<T>> mycont; //使用默认模板参数，可以省略第二个参数
```

例如，下列语句定义了两个 list 容器：

```
class TMyClass;
typedef list<TMyClass> TMyClassList; // 用于存放对象的 list 容器
typedef list<TMyClass*> TMyClassPtrList; // 用于存放对象指针的 list 容器
```

其包含的成员函数如下。

- **resize**: 被控序列的长度改为只容纳 n 个元素，超出的元素将被删除，如果需要扩展，那么默认值为 $T()$ 的元素会被放到序列末端。
- **clear**: 删除所有元素。
- **front, back**: 存取被控序列的第一个元素；存取被控序列的最后一个元素。如果它不是一个常量表达式可以作为一个左值。如果序列为空，则这种行为将是未定义的。
- **push_back**: 向对象末端插入值为 x 的元素。**push_front** 为对象开始处插入元素；**pop_back** 删除最后一个元素；**pop_front** 删除第一个元素；序列必须不为空，否则调用将是未定义的。
- **assign**: 为了将被控序列替换成由 $(first, last)$ 所指定的序列，且不能是原序列的一部分。
- **insert**: 为了在迭代器 it 指定的元素前插入一个元素，返回值是一个迭代器，指向刚插入的元素，也可以插入一个序列。
- **erase**: 删除 it 所指定的元素，返回值为一个迭代器，指向下一个元素，也可以删除一



个区间。

- **splice**: 将一系列的列表节点接入到一个列表中, 被接入的节点列表将会被删除, 接入操作中没有元素复制, 只是将节点中的指针改写; `cont.splice(it,cont2)`把对象 `cont2` 中所有内容接入, 这两个对象必须不同; `cont.splice(it,cont2,p)`将对象 `cont2` 中迭代器 `p` 指定的节点接入到由迭代器 `it` 指定的节点前面, 这两个对象可以相同; `cont.splice(it,cont2,first,last)`将 `cont2` 中指定的序列接入到迭代器 `it` 所指定的元素前面, 这两个对象可以相同, 但 `it` 不能是序列的一部分; 这两个列表所拥有的分配器对象必须相等。
- **remove**: 删除所有值等于 `v` 的元素; `remove_if` 删除所有 `pr(x)`为 `true` 的元素 `x`; 在 STL 中比较特殊的, 它真正地删除了元素 `-b-`。
- **unique**: 删除指定范围内相同的元素, 在 STL 中比较特殊, 好像 `-#`。
- **sort**: 将序列排序, 结果序列是按 `operator<`排序的, 操作中用到了接入操作, 可以将 `pr` 作为排序函数。
- **merge**: 将两个有序排序序列合并, 合并中用到了接入操作, 合并后第二个序列将为空, 可以用 `pr` 替换排序函数。
- **reverse**: 翻转整个序列。

【范例 18-4】使用列表容器。该范例使用列表容器存储数据内容, 进行排序后输出, 实现代码如代码清单 18-4 所示。

代码清单 18-4

```

1  #include <list>                                //包含列表类头文件
2  #include <iostream>
3  void main()
4  {
5      using namespace std;                        //使用命名空间
6      list<int> c1;                                //定义列表变量
7      list<int>::iterator c1_iter;
8      c1.push_back(20);                            //调用函数写入列表
9      c1.push_back(10);
10     c1.push_back(30);
11     cout<<"Before sorting: c1 =";
12     for ( c1_iter = c1.begin( ); c1_iter != c1.end( ); c1_iter++ )
13         cout << " " << *c1_iter;                //输出列表中内容
14     cout << endl;
15     c1.sort( );                                    //调用列表类排序函数
16     cout << "After sorting c1 =";
17     for ( c1_iter = c1.begin( ); c1_iter != c1.end( ); c1_iter++ )
18         cout << " " << *c1_iter;                //输出列表中内容
19     cout << endl;
20     c1.sort( greater<int>( ) );                    //调用降序排列函数
21     cout << "After sorting with 'greater than' operation, c1 =";
22     for ( c1_iter = c1.begin( ); c1_iter != c1.end( ); c1_iter++ )
23         cout << " " << *c1_iter;                //输出列表中内容
24     cout << endl;
25 }
```

【运行结果】在 Visual C++中新建一个【C++ Source file】, 输入如上的代码, 编译运行无误后, 其结果如图 18-6 所示。

【范例解析】上述代码中, 首先调用函数 `push_back` 向列表中写入三个数值并输出, 接着分别对其进行升序和降序的排序, 分别调用不带参数的 `sort` 函数和带参数的 `sort` 函数实现, 并将结果输出。读者可比较其不同的结果, 观察 `list` 容器内的函数功能。

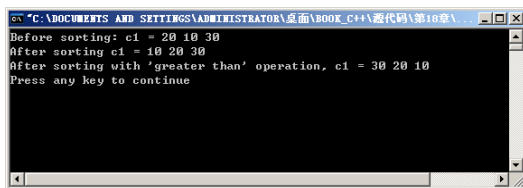


图 18-6 使用列表容器

18.3.4 集合

集合(set)也是容器的一种,它的特点是在集合中的元素值是唯一的。在集合中,所有的成员都是排列好的。如果先后往一个集中插入:12,2,3,123,5,65,则输出该集时为:2,3,5,12,65,123。



注意 集和多集的区别是: set 支持唯一键值, set 中的值都是特定的, 而且只出现一次; 而 multiset 中可以出现副本键, 同一值可以出现多次。

【范例 18-5】使用集合。该范例使用集合存储数据内容, 将值插入到集合后输出, 实现代码如代码清单 18-5 所示。

代码清单 18-5

```

1  #include <string>
2  #include <set>                                //包含集合类头文件
3  #include <iostream>
4  using namespace std;                          //使用命名空间
5  int main(int argc, char* argv[])
6  {
7      set<string> strset;                        //定义变量
8      set<string>::iterator si;
9      strset.insert("cantaloupes");             //调用函数插入数据到集合中
10     strset.insert("apple");
11     strset.insert("orange");
12     strset.insert("banana");
13     strset.insert("grapes");
14     strset.insert("grapes");
15     for (si=strset.begin(); si!=strset.end(); si++)
16     {                                           //循环输出集合中数据
17         cout<< *si << " ";                  //输出值
18     }
19     cout << endl;                             //换行
20     return 0;
21 }
```

【运行结果】在 Visual C++ 中新建一个【C++ Source file】, 输入如上的代码, 编译运行无误后, 其结果如图 18-7 所示。

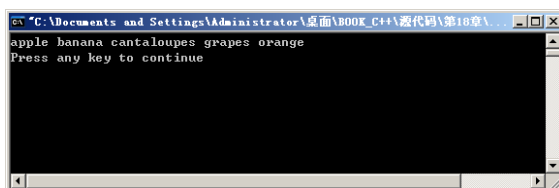


图 18-7 使用集合



【范例解析】上述代码实现向字符串集合中插入几个字符串，读者可以看到，元素被插入到集合后，使用 for 语句将其输出，其结果为已经排好序的、删除了重复元素值的字符串元素序列，这就是集合容器的特征。

18.3.5 双端队列

双端队列是一个 queue 容器类（队列容器），其定义在头文件 deque 中（double queue）。在使用该容器时，需在头文件处加上如下语句：

```
#include <deque>
```

双端队列容器类与 vector 类似，支持随机访问和快速插入删除，它在容器中某一位置上的操作所花费的是线性时间。与 vector 不同的是，deque 还支持从开始端插入数据，因为其包含在开始端插入数据的函数 push_front()。此外 deque 也不支持与 vector 的 capacity()、reserve() 类似的操作。

【范例 18-6】使用双端队列。该范例使用双端队列存储数据内容，对该队列进行插入和删除元素操作，实现代码如代码清单 18-6 所示。

代码清单 18-6

```
1  #include <iostream>
2  #include <deque>
3  using namespace std;
4  typedef deque<int> INTDEQUE;           //定义双端队列容器
5  void put_deque(INTDEQUE deque, char *name) //从前向后显示 deque 队列的全部元素
6  {
7      INTDEQUE::iterator pdeque;         //仍然使用迭代器输出
8      cout << "The contents of " << name << " : ";
9      for(pdeque = deque.begin(); pdeque != deque.end(); pdeque++)
10         cout << *pdeque << " ";        //注意有 "+" 号，否则会报错
11     cout<<endl;
12 }
13 void main(void)                         //测试 deqtor 容器的功能
14 {
15
16     INTDEQUE deq1;                       //定义双端队列 deq1 对象初始为空
17     INTDEQUE deq2(10,6);                 //deq2 对象最初有 10 个值为 6 的元素
18     INTDEQUE::iterator i;               //声明一个名为 i 的双向迭代器变量
19     put_deque(deq1,"deq1");              //从前向后显示 deq1 中的数据
20     put_deque(deq2,"deq2");              //从前向后显示 deq2 中的数据
21     deq1.push_back(2);                   //从 deq1 序列后面添加两个元素
22     deq1.push_back(4);
23     cout<<"deq1.push_back(2) and deq1.push_back(4):"<<endl;
24     put_deque(deq1,"deq1");              //显示 deq1 中的元素
25     deq1.push_front(5);                  //从 deq1 序列前面添加两个元素
26     deq1.push_front(7);
27     cout<<"deq1.push_front(5) and deq1.push_front(7):"<<endl;
28     put_deque(deq1,"deq1");
29     deq1.insert(deq1.begin()+1,3,9);     //在 deq1 序列中间插入数据
30     cout<<"deq1.insert(deq1.begin()+1,3,9):"<<endl;
31     put_deque(deq1,"deq1");
32     cout<<"deq1.at(4) ="<<deq1.at(4)<<endl; //测试引用类函数
33     cout<<"deq1[4] ="<<deq1[4]<<endl;
34     deq1.at(1)=10;
35     deq1[2]=12;
36     cout<<"deq1.at(1)=10 and deq1[2]=12 : "<<endl;
```

```

37     put_deque(deq1,"deq1");           //显示 deq1 的元素
38     deq1.pop_front();                 //从 deq1 序列的前后各移去一个元素
39     deq1.pop_back();
40     cout<<"deq1.pop_front() and deq1.pop_back():"<<endl;
41     put_deque(deq1,"deq1");
42     deq1.erase(deq1.begin()+1);        //清除 deq1 中的第 2 个元素
43     cout<<"deq1.erase(deq1.begin()+1):"<<endl;
44     put_deque(deq1,"deq1");
45     deq2.assign(8,1);                  //对 deq2 赋值并显示
46     cout<<"deq2.assign(8,1):"<<endl;
47     put_deque(deq2,"deq2");
48 }

```

【运行结果】在 Visual C++ 中新建一个【C++ Source file】，输入如上的代码，编译运行无误后，其结果如图 18-8 所示。

【范例解析】上述代码演示了 deque 如何进行插入删除等操作，双端队列 deq1 首先为空，通过函数 push_front() 和函数 push_back() 分别在该队列的前端和后端插入元素，通过函数 insert() 在队列的中间插入元素，通过函数 erase() 在队列中删除元素。

```

C:\DOCUMENTS AND SETTINGS\ADMINISTRATOR\桌面\BOOK C++\源代码\第18章\
The contents of deq1 :
The contents of deq2 : 6 6 6 6 6 6 6 6
deq1.push_back(2) and deq1.push_back(4):
The contents of deq1 : 2 4
deq1.push_front(5) and deq1.push_front(7):
The contents of deq1 : 7 5 2 4
deq1.insert(deq1.begin()+1,3,9):
The contents of deq1 : 7 9 9 5 2 4
deq1.at(4)=5
deq1[4]=5
deq1.at(1)=10 and deq1[2]=12 :
The contents of deq1 : 7 10 12 9 5 2 4
deq1.pop_front() and deq1.pop_back():
The contents of deq1 : 10 12 9 5 2
deq1.erase(deq1.begin()+1):
The contents of deq1 : 10 9 5 2
deq2.assign(8,1):
The contents of deq2 : 1 1 1 1 1 1 1 1
Press any key to continue_

```

图 18-8 使用双端队列



提示 诸如函数 erase() 和 assign() 等是大多数容器都有的操作，其分别表示删除容器中的元素和给容器赋值。

18.3.6 栈

容器栈 (stack) 是一种特殊的容器，其特征是后进先出，即先进来的元素放在栈底，最后才能取出。栈容器支持的操作有如下 5 种。

- empty(): 如果栈为空，则返回 true，否则返回 false。
- size(): 返回栈中元素的个数。
- pop(): 删除，但不返回栈顶元素。
- top(): 返回，但不删除栈顶元素。
- push(item): 放入新的栈顶元素。

对栈容器的操作都是通过上述的 5 个基本函数来实现的。

【范例 18-7】使用栈容器。该范例实现了将 10 个元素放入栈，然后输出，读者观察其输出是否有变化，实现代码如代码清单 18-7 所示。

代码清单 18-7

```

1  #include <stack>                      //包含栈类头文件

```



```

2  #include <iostream>
3  using namespace std;                                //使用命名空间
4  int main(int argc, char* argv[])
5  {
6      const int ia_size = 10;                          //定义常量
7      int ia[ia_size] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; //定义数组
8      int ix = 0;
9      stack<int> intStack;                             //定义栈类对象
10     for ( ; ix < ia_size; ++ ix)
11     {
12         intStack.push(ia[ix]);                       //调用栈类函数压入数组元素
13     }
14     if (intStack.size() != ia_size)                  //判断栈的大小
15     {
16         cout << "error!" << endl;
17         return -1;
18     }
19     int value;
20     while (!intStack.empty())                        //循环输出栈内元素
21     {
22         value = intStack.top();                      //取栈顶元素
23         cout<<value<<" ";
24         intStack.pop();                             //删除该栈顶元素
25     }
26     cout<<endl;
27     return 0;
28 }

```

【运行结果】在 Visual C++ 中新建一个【C++ Source file】，输入如上的代码，编译运行无误后，其结果如图 18-9 所示。

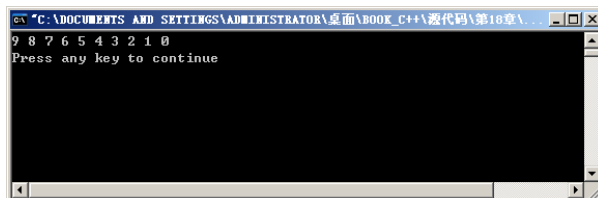


图 18-9 使用栈容器

【范例解析】上述代码首先定义了一个包含 10 个元素的数组，通过 for 循环语句将数组中的元素依次使用栈的函数 push() 将其放入栈中，再通过 while 语句将该栈中的元素依次取出，使用的取出函数为 top()。

读者可以看出，上述代码返回的结果是生成了数组元素的倒序。由于栈的特征是后进先出，最后放入栈的元素是数组的最后一个元素 9 位于栈顶，因此在输出时，该元素最先输出，其他的元素输出以此类推。



注意 stack 只是很单纯地把各项操作转化为内部容器的对应调用，读者可以使用任何序列式容器来支持 stack，只要其支持 back()、push_back()、pop_back() 等动作即可。

18.3.7 映射和多重映射

映射和多重映射用于对数据进行快速和高效的检索。同样，在程序中使用映射和多重映射容器需添加如下头文件：

```
#include <map>
```

此外,映射 map 支持下标运算符 operator[], 可以用访问普通数组的方式访问 map, 或下标为 map 的键。而在 multimap 中一个键可以对应多个不同的值。

【范例 18-8】使用映射容器。该范例使用映射容器存储数据内容, 并调用其函数完成元素键值的检索功能, 实现代码如代码清单 18-8 所示。

代码清单 18-8

```

1  #include <iostream>
2  #include <map>                                //包含映射头文件
3  using namespace std;                          //使用命名空间
4  int main(void)
5  {
6      map<char,int,less<char> > map1;           //定义变量
7      map<char,int,less<char> >::iterator mapIter;
                                                //char 是键的类型, int 是值的类型
8      map1['c']=3;
9      //初始化, 与数组类似也可以用 map1.insert(map<char,int,less<char> >::value_type
      // ('c',3));
10     map1['d']=4;
11     map1['a']=1;
12     map1['b']=2;
13     for(mapIter=map1.begin();mapIter!=map1.end();++mapIter)
14         cout<<" "<<(*mapIter).first<<": "<<(*mapIter).second;
15     //first 对应定义中的 char 键, second 对应定义中的 int 值
16     cout<<endl;
17     map<char,int,less<char> >::const_iterator ptr;
18     ptr=map1.find('d');                        //检索对应于 d 键的值
19     cout<<" "<<" "<<(*ptr).first<<"键对应于值: "<<(*ptr).second<<endl;;
                                                //输出
20     return 0;
21 }
```

【运行结果】在 Visual C++ 中新建一个 **【C++ Source file】**, 输入如上的代码, 编译运行无误后, 其结果如图 18-10 所示。

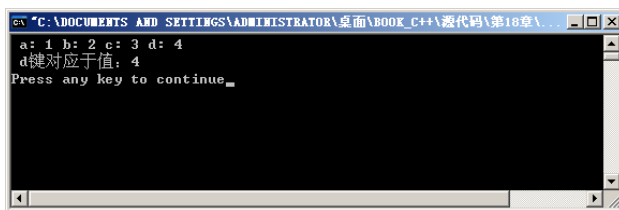


图 18-10 使用映射容器

【范例解析】上述的范例说明了 map 中键与值的关系。在上述代码中, 首先定义了映射容器, 其键的类型为字符型, 值的类型为整型。对该容器 map1 进行了初始化, 在其中存储 4 个字符对应的整型数值, 并输出。再通过函数 find() 找出其中键 d 对应的值。



提示 在 map 中是不允许一个键对应多个值的, 而在多重映射 multimap 中, 不支持 operator[], 即它不支持 map 中允许的下标操作。

18.4 迭代器

迭代器实际上是一种泛化指针, 如果一个迭代器指向了容器中的某一成员, 那么迭代器将



可以通过自增自减来遍历容器中的所有成员。迭代器是联系容器和算法的媒介，是算法操作容器的接口，如图 18-11 所示。

简单来说，几乎 STL 提供的所有算法都是通过迭代器存取元素序列进行工作的，每一个容器都定义了它本身所专有的迭代器，用以存取容器中的元素。在前面运用算法操作容器的时候，就在不知不觉中已经使用了迭代器，如代码清单 18-6 中的语句 `INTDEQUE::iterator pdeque;` 即表示通过迭代器来输出。



图 18-11 迭代器的作用

STL 中的迭代器主要由头文件 `<utility>`、`<iterator>` 和 `<memory>` 组成。其中，`<utility>` 包括了贯穿使用在 STL 中的几个模板的声明，`<iterator>` 头文件中提供了迭代器使用的许多方法。`<memory>` 头文件中的主要部分是模板类 `allocator`，它负责产生所有容器中的默认分配器。

【范例 18-9】使用输入/输出迭代器。该范例使用输入/输出迭代器完成了将一个文件输出到屏幕的功能，实现代码如代码清单 18-9 所示。

代码清单 18-9

```

1  #include <iostream>
2  #include <fstream>           //包含文件输入/输出头文件
3  #include <iterator>          //包含迭代器头文件
4  #include <vector>            //包含向量容器类头文件
5  #include <string>
6  using namespace std;        //使用命名空间
7  int main(void)
8  {
9      vector<string> v1;       //定义向量对象
10     ifstream file("test.txt"); //打开当前文件夹下的 test.txt 文件
11     if(file.fail())          //打开失败
12     {
13         cout<<"open file Text1.txt failed"<<endl;
14         return 1;
15     }
16     copy(istream_iterator<string>(file), istream_iterator<string>(), inserter
(v1, v1.begin()));
17     copy(v1.begin(), v1.end(), ostream_iterator<string>(cout, " "));
                                //复制内容
18     cout<<endl;              //输出换行
19     return 0;
20 }
  
```

【运行结果】由于该范例中使用到了文件输入流，需要打开文件 `test.txt`，因此需在当前文件夹下新建一个文本文件 `test.txt`，在该文本文件中输入字符串“Welcome to Visual C++”，此时再查看该范例的运行效果，如图 18-12 所示。

【范例解析】上述范例实现从文件中读取内容并将其输出在屏幕上。与前面文件流的操作不同的是，上述代码使用了文件的输入/输出迭代器来实现。上述代码中，用到了输入迭代器 `istream_iterator`，输出迭代器 `ostream_iterator`。程序完成了将一个文件输出到屏幕的功能，先将文件读入，然后通过输入迭代器把文件内容复制到类型为字符串的向量容器内，最后由输出迭代器输出。`inserter` 是一个输入迭代器的函数（迭代器适配器），其使用方法是：

```
inserter (container ,pos);
```

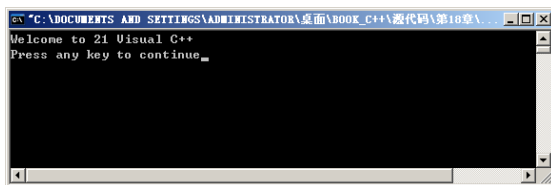


图 18-12 使用输入/输出迭代器



注意

上述代码中, container 是将要用来存入数据的容器, pos 是容器存入数据的开始位置。上述范例中, 是把文件内容通过函数 copy() 存入到向量 v1 中。

18.5 小结

本章主要介绍了标准模板库 STL 的相关内容。首先对 STL 的概念及其在程序设计中的重要性做了概括介绍, 并通过一个具体示例引入了 STL 的应用。本章主要讲解了 STL 的几个组成部分, 包括算法、容器、迭代器等, 对于每个组成部分在具体程序中的使用, 都通过一个实例来讲解, 读者仔细理解这些实例即可理解 STL 的优势。

18.6 习题

1. 将包含 10 个元素的序列逐个插入 vector 容器中, 插入完成后将 vector 容器中的所有元素输出。

【解答】该习题主要考查 vector 容器的使用。根据前面的学习, 读者知道 vector 容器利用索引直接存取任何一个元素, 在尾部插入元素或删除元素均非常快速。此处调用 vector 容器的一些函数即可, 需要读者注意的是, 在头文件中必须加上 vector。其简要的实现代码如下所示。

```
#include <vector>
int main(){
    vector<int> coll;
    for (i = 1; i <= 6; ++i){
        coll.push_back(i);
    }
    for (i = 0; i < coll.size(); ++i){
        cout << coll[i] << " ";
    }
}
```

2. 编写一个 C++ 程序, 创建有 10 个元素的 vector 对象, 并使用迭代器把其中的每个元素改为当前值的 2 倍并输出。

【解答】该习题主要考查 vector 容器和迭代器的使用。读者知道, 迭代器实际上是一种泛化指针, 如果一个迭代器指向了容器中的某一成员, 那么迭代器将可以通过自增自减来遍历容器中的所有成员。在该习题中, 可以通过迭代器依次对 vector 容器中的元素进行乘以 2 的操作。其简要的实现代码如下所示。

```
vector<int> ivec;

for(vector<int>::size_type ix=0;ix!=10;++ix){
    ivec.push_back(ix);
    cout<<ivec[ix]<<'\t';
}

for(vector<int>::iterator iter=ivec.begin();iter!=ivec.end();++iter){
    *iter*=2;
    cout<<*iter<<'\t';
}
```



```
}

```

上述程序中,首先定义 vector 容器对象 ivec,再使用 for 循环语句将 10 个元素存入容器中,此处使用了 push_back 函数,实现从尾部进行元素插入,最后使用循环语句通过迭代器依次对其中的元素进行乘以 2 的操作并同时输出。

3. 下面程序的输出结果是什么?

```
#include <set>
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
    set <string> strset;
    set <string>::iterator si;
    strset.insert("cantaloupes");
    strset.insert("apple");
    strset.insert("orange");
    strset.insert("banana");
    strset.insert("grapes");
    strset.insert("grapes");
    for (si=strset.begin(); si!=strset.end(); si++)
        { cout << *si << " "; =
    return 0;
}
```

【解答】该习题主要考查集合容器的使用。根据前面的学习,读者知道集合的特点是其中的元素值是唯一的,且所有的成员都是排列好的。因此,在上述程序中,输出的集合中的元素是按字母大小顺序排列的,而且每个值都不重复。因此,该程序段的输出为:

```
apple banana cantaloupes grapes orange
```

4. 编写一个 C++ 程序,创建了一个矢量容器 (STL 的和数组等价的对象),并使用迭代器在其中搜索,找到值为 100 的元素。

【解答】该习题主要考查迭代器在容器中的使用。该习题可以使用 vector 容器来实现存储数据,其相当于数组,通过迭代器的 begin()和 end()方法表示容器的首尾,通过循环语句依次对容器中的所有元素进行比较,直到找到指定元素为止。其简要的实现代码如下所示。

```
#include <iostream.h>
#include <algorithm>
vector<int> intVector(100);
intVector[20] = 100;
vector<int>::iterator intIter =
    find(intVector.begin(), intVector.end(), 100);
if (intIter != intVector.end())
    cout << "容器包含元素 100" << *intIter << endl;
else
    cout << "容器不包含元素 100" << endl;
```

5. 编写一个 C++ 程序,求出范围在 2~N 之间的所有素数,其中 N 在程序运行时由键盘输入。例如,给出 N 的值为 15 时,其输出结果如图 18-13 所示。

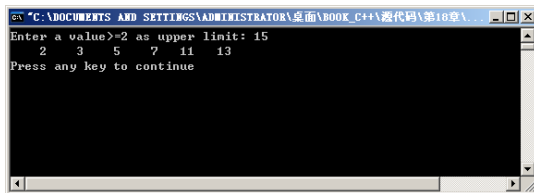


图 18-13 求素数

【解答】该程序段可使用向量容器，用于存储素数，同时采用循环取余的方法求出范围在 2~N 之间的所有素数。其简要的实现代码如下所示。

```
#include <iostream>
#include <iomanip> //包含 I/O 流控制头文件
#include <vector> //包含向量容器头文件
using namespace std; //使用存储过程
void main() //主函数
{
    vector<int> A(10); //定义向量对象
    int n; //定义整型变量
    int primecount = 0, i, j; //定义变量并初始化
    cout<<"Enter a value>=2 as upper limit: "; //输入提示
    cin >> n; //接收 n 的输入
    A[primecount++] = 2; //赋值
    for(i = 3; i < n; i++) //循环输出素数
    {
        if (primecount == A.size()) //到达向量大小
            A.resize(primecount + 10); //调用函数重新设置向量大小
        if (i % 2 == 0) //不是素数
            continue; //继续循环
        j = 3; //赋初值
        while (j <= i/2 && i % j != 0) //循环判断是否为素数
            j += 2; //变量递增 2
        if (j > i/2) //是素数
            A[primecount++] = i; //写入向量中
    }
    for (i = 0; i<primecount; i++) //输出素数
    {
        cout<<setw(5)<<A[i]; //设置输出格式并输出
        if ((i+1) % 10 == 0) //每输出 10 个数换行一次
            cout << endl; //输出换行
    }
    cout<<endl; //输出换行
}
```

第 19 章 模板与 C++ 标准库

有程序设计语言基础的读者应该听说过模板的概念，事实上，在前面的章节中，就已经使用到了模板。模板是实现代码复用的一种工具，其有函数模板和类模板之分。此外，模板是现代 C++ 程序设计中的一个重要概念，使用模板可大大减少代码数量，提高代码的效率。第 18 章介绍了标准模板库 STL，读者知道了 C++ 还包含许多可直接调用的组件和函数。事实上，C++ 还包含了标准库，用于提供符合 C++ 标准的函数。本章将就模板和 C++ 标准库为读者做简要介绍，使读者对这两个概念有一定的理解。

以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为一个学习的参照。

- 理解模板的概念。
- 掌握函数模板和类模板的定义和生成。
- 理解 C++ 标准库及字符串库。

19.1 模板概述

C++ 模板是近几年来对 C++ 的一种扩展，模板是根据类型参数来产生函数和类的机制。使用模板可以设计一个对许多类型的数据进行操作的类，而不需要为每个类型的数据建立一个单独的类。第 18 章所介绍的标准模板库就是基于 C++ 的模板之一。

19.1.1 模板简介

简单地说，模板是实现代码复用的一种工具，它可以实现类型参数化，把类型定义为参数，实现代码的真正复用。

事实上，模板是 C++ 在 20 世纪 90 年代引进的一个新概念，原本是为了对容器类（container class）的支持，但是现在模板产生的效果已经远非当初所能想象。或者说，模板就是一种参数化的类或函数，即类的形态（成员、方法、布局等）或者函数的形态（参数、返回值等）可以被参数改变。此外，此处所说的参数，不光是传统函数中所说的数值形式的参数，还可以是一种类型。

例如，在 C 语言中，如果要比较两个数的大小，常常会定义两个宏，通过宏来实现大小比较，并在函数中调用宏即可，宏定义如下：

```
#define min(a,b) ((a)>(b)?(b):(a))
#define max(a,b) ((a)>(b)?(a):(b))
```

定义后，读者就可以在程序中通过如下的代码调用：

```
return min(10, 4);
```

或者

```
return min(5.3, 18.6);
```

这两个宏非常好用，但是在 C++ 中，它们并不像在 C 中那样受欢迎。宏因为没有类型检查，而且天生的不安全，因此在 C++ 中被 inline 函数替代。但是随着将 min() 和 max() 改为函数，用户立刻就会发现这个函数的局限性，即它们不能处理用户指定的类型以外的其他类型，如 min() 声明为：

```
int min(int a, int b);
```

它显然不能处理 `float` 类型的参数，但是原来的宏却可以很好地工作。读者随后大概会想到函数重载，通过重载不同类型的 `min()` 函数，仍然可以使大部分代码正常工作。实际上，C++ 对于这类可以抽象的算法，提供了更好的办法，这就是模板。

19.1.2 模板的引入

在前面章节 C++ 语言宏的定义中，读者看到了宏的不足，在 C++ 中，提供了模板的概念用于解决这类问题。例如，可以将上述 `min()` 和 `max()` 函数定义为模板，如下所示：

```
template <class T> const T & min(const T & t1, const T & t2)
{
    return t1>t2?t2:t1;
}
template <class T> const T & max(const T & t1, const T & t2)
{
    return t1>t2?t1:t2;
}
```

这是一个模板函数 `min` 和 `max` 的例子。在有了模板之后，用户就可以像原来在 C 语言中使用 `min` 宏一样来使用这个模板，例如：

```
return min(10,4);
```

或者

```
return min(5.3, 18.6)
```



提示 读者可以发现，通过定义模板，用户获得了一个类型安全、而又可以支持任意类型的 `min()` 和 `max()` 函数。

【范例 19-1】模板的引入。该范例使用了模板来实现多个数据类型的求最大最小值，实现代码如代码清单 19-1 所示。

代码清单 19-1

1	#include <iostream.h>	//包含输入/输出头文件
2	template<typename T>	//定义模板
3	const T & min(const T & t1, const T & t2)	
4	{	
5	return t1>t2?t2:t1;	//返回较小值
6	}	
7	template<typename T>	//定义模板
8	const T & max(const T & t1, const T & t2)	
9	{	
10	return t1>t2?t1:t2;	//返回较大值
11	}	
12	void main()	
13	{	
14	int a,b;	//定义整型变量
15	int mn,mx;	
16	cout<<"Please input 2 numbers:"<<endl;	
17	cin>>a>>b;	//接收用户输入
18	mn=min(a,b);	//调用函数
19	cout<<"The Min is :"<<mn<<endl;	
20	mx=max(a,b);	//调用函数
21	cout<<"The Max is :"<<mx<<endl;	
22	}	



【运行结果】在 Visual C++ 中新建一个【C++ Source File】文件，输入上述的代码，编译无误后运行，其结果如图 19-1 所示。

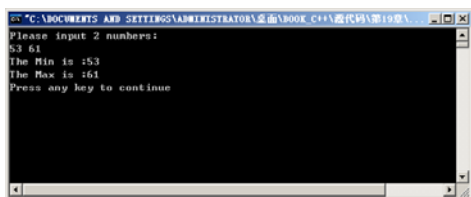


图 19-1 模板的引入

【范例解析】上述代码中，第 2~6 行定义了求最小值的模板函数 `min()`，在第 7~11 行中定义了求最大值的模板函数 `max()`，在主函数中分别调用了这两个模板函数。事实上，模板的作用远不只是用来替代宏。



注意 实际上，模板是泛化编程的基础。所谓的泛化编程，就是对抽象的算法的编程，泛化指可以广泛适用于不同的数据类型，如上面提到的 `min()` 和 `max()` 函数。

19.2 函数模板

19.1 节提到了模板分两类：函数模板和类模板，用户可使用它们来构造模板函数或模板类。模板经过实例化后就得到模板函数或模板类，模板函数或模板类在经过实例化后就得到对象。本节首先介绍函数模板。

19.2.1 定义函数模板

函数模板可以用来创建一个通用的函数，以支持多种不同的形参，避免重载函数的函数体重复设计，其最大特点是把函数使用的数据类型作为参数。一般来说，函数模板的定义形式为：

```
template<typename 数据类型参数标识符>
<返回类型><函数名>(参数表)
{
    函数体
}
```

其中，`template` 是定义模板函数的关键字，它后面的尖括号不能省略；`typename` 是声明数据类型参数标识符的关键字，用以说明它后面的标识符是数据类型标识符。

这样，在以后定义的这个函数中，凡希望根据实参数数据类型来确定数据类型的变量都可以用数据类型参数标识符来说明，从而使这个变量可以适应不同的数据类型。例如：

```
template<typename T>
T fuc(T x, int y)
{
    T x;
    //.....
}
```

此外，关键字 `typename` 也可以使用关键字 `class`，这时数据类型参数标识符就可以使用所有的 C++ 数据类型。

如果主调函数中有以下语句：

```
double d;
int a;
fuc(d,a);
```

则系统将用实参 `d` 的数据类型 `double` 去代替函数模板中的 `T` 生成函数，即将上述定义的函数模板变成如下的形式：

```
double fuc(double x,int y)
{
```

```
double x;
//.....
}
```



函数模板只是声明了一个函数的描述 (即模板), 不是一个可以直接执行的函数, 只有根据实际情况用实参的数据类型代替类型参数标识符之后, 才能产生真正的函数。

19.2.2 生成模板函数

19.2.1 节提到了, 函数模板只是一个模板, 并不是真正的函数。函数模板的数据类型参数标识符实际上是一个类型形参, 在使用函数模板时, 要将这个形参实例化为确定的数据类型。将类型形参实例化的参数称为模板实参, 用模板实参实例化的函数称为模板函数。模板函数的生成就是将函数模板的类型形参实例化的过程。

同样, 采用范例 19-1 中定义的函数模板, 在 19-1 的主程序中将它实例化为整型数据类型, 即生成了整型模板函数。

【范例 19-2】生成模板函数。该范例将上述模板函数实例化为浮点型的模板函数, 代码如代码清单 19-2 所示。

代码清单 19-2

```
1  #include <iostream.h>                                //包含输入/输出头文件
2  template<typename T>                                  //定义模板
3  const T & min(const T & t1, const T & t2)
4  {
5      return t1>t2?t2:t1;                                //返回较小值
6  }
7  template<typename T>                                  //定义模板
8  const T & max(const T & t1, const T & t2)
9  {
10     return t1>t2?t1:t2;                                //返回较大值
11 }
12 void main()
13 {
14     double a,b;                                         //定义浮点型变量
15     double mn,mx;
16     cout<<"Please input 2 numbers:"<<endl;
17     cin>>a>>b;                                         //接收用户输入
18     mn=min(a,b);                                       //调用函数
19     cout<<"The Min is : "<<mn<<endl;
20     mx=max(a,b);                                       //调用函数
21     cout<<"The Max is : "<<mx<<endl;
22 }
```

【运行结果】同样, 在 Visual C++ 中新建一个 **【C++ Source File】** 文件, 输入上述的代码, 编译无误后运行, 其结果如图 19-2 所示。

【范例解析】读者可以看出, 上述代码与代码 19-1 的区别就在于其定义的变量 a、b 为浮点型数据类型, 而不是整型。此外, 函数模板并没有做任何改变, 调用后的函数 min() 和 max() 就可以对两个浮点型数据进行比较。

因此, 使用函数模板来生成模板函数, 在实际的程序中是非常方便的。但是, 在使用模板函数时候, 需要注意如下的几个事项。

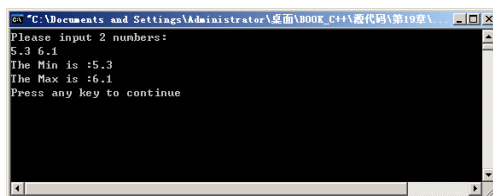


图 19-2 生成模板函数



- 函数模板允许使用多个类型参数，但在 `template` 定义部分的每个形参前必须有关键字 `typename` 或 `class`，即：

```
template<class 数据类型参数标识符 1, ..., class 数据类型参数标识符 n>
<返回类型><函数名>(参数表)
{
    函数体
}
```

- 在 `template` 语句与函数模板定义语句<返回类型>之间不允许有别的语句。如下面的声明是错误的：

```
template<class T>
int I;
T min(T x, T y)
{
    函数体
}
```



模板函数类似于重载函数，但两者区别很大。函数重载时，每个函数体内可以执行不同的动作，但同一个函数模板实例化后的模板函数都必须执行相同的动作。

19.2.3 函数模板的异常处理

函数模板中的模板形参可实例化为各种类型，但当实例化模板形参的各模板实参之间不完全一致时，就可能发生错误。例如：

```
template<typename T>
void min(T &x, T &y)
{
    return (x<y)?x:y;
}
void func(int i, char j)
{
    min(i, i);
    min(j, j);
    min(i, j);
    min(j, i);
}
```

上述例子中的后两个调用是错误的，出现错误的原因是，在调用时，编译器按最先遇到的实参的类型隐含地生成一个模板函数，并用它对所有模板函数进行一致性检查，如语句：

```
min(i, j);
```

先遇到的实参 `i` 是整型的，编译器就将模板形参解释为整型，而此后出现的模板实参 `j` 不能解释为整型这样就产生错误，因为没有隐含的类型转换功能。解决该异常的方法有两种：

- 采用强制类型转换，如将语句 `min(i, j);` 改写为 `min(i, int(j));`。
- 用非模板函数重载函数模板。

其中，采用非模板函数重载函数模板的方法有两种：一种是直接使用函数模板的函数体，另一种是重新定义函数体。如果采用直接使用函数模板的函数体，则只需声明非模板函数的原型，它的函数体借用函数模板的函数体。将上面的例子改写如下：

```
template<typename T>
void min(T &x, T &y)
{
    return (x<y)?x:y;
}
```

```
int min(int,int);
void func(int i, char j)
{
    min(i, i);
    min(j, j);
    min(i, j);
    min(j, i);
}
```

这样执行该程序就不会出错了,因为重载函数支持数据间的隐式类型转换。此外,还可以采用重新定义函数体的方法解决这个问题。



提示 就像一般的重载函数一样,重新定义一个完整的非模板函数,它所带的参数可以随意。C++中,函数模板与同名的非模板函数重载时,应遵循下列调用原则:

- 寻找一个参数完全匹配的函数,若找到就调用它。若参数完全匹配的函数多于一个,则这个调用是一个错误的调用。
- 寻找一个函数模板,将它实例化生成一个匹配的模板函数,若找到就调用它。
- 若上面两项都失败,再使用第一级的对函数重载的方法,如通过类型转换产生参数匹配,若找到就调用它。
- 若上面三项都失败,依然还没有找到匹配的函数,则这个调用是一个错误的调用。

19.3 类模板

19.2 节介绍了函数模板及将函数模板进行实例化后形成的模板函数,本节将讲解第二种模板:类模板。

19.3.1 定义类模板

类模板也称为类属类或类生成类,是为类定义的一种模式,它使类中的一些数据成员和成员函数的参数或返回值可以取任意的数据类型。类模板是一个具体的类,它代表一族类,是这一族类的统一模式,使用类模板就是要将它实例化为具体的类。

一般来说,定义类模板的一般形式为:

```
template<class 数据类型参数标识符>
class 类名
{
    //.....
}
```

其中, **template** 是声明类模板的关键字,它后面的尖括号不能省略;数据类型参数标识符是类模板中参数化的类型名,当实例化类模板时,它将由一个具体的类型来代替。



注意 定义类模板时,可以声明多个类型参数标识符,各标识符之间用逗号分开。

类定义中,凡要采用标准数据类型的数据成员、成员函数的参数或返回类型的前面都要加上类型标识符。如果类中的成员函数要在类的声明之外定义,则它必须是模板函数。其定义形式为:

```
template<class 数据类型参数标识符>
数据类型参数标识符 类名<数据类型参数标识符>::函数名(数据类型参数标识符 形参1,...,数据类型参数标识符 形参n)
{
    函数体
}
```



```
}

```

例如，下列语句定义了一个类模板 `Stack`，该模板是一个栈模板，它包含了私有成员和公有成员，其中还包含了出栈和入栈函数。

```
template<typename T>
class Stack
{
public:
    Stack(int size = 10);
    ~Stack();
    bool Push(const T&);
    bool Pop(T &);
private:
    int iSize;
    int iTop;
    T *stackPtr;
    bool isEmpty()const;
    bool isFull()const;
};

```

19.3.2 模板类

与模板函数的生成相似，将类模板的模板参数实例化后生成的具体的类，就是模板类。由类模板生成模板类的一般形式为：

类名<数据类型参数标识符>对象名 1, 对象名 2, ..., 对象名 n;

此处的数据类型参数标识符对应的是对象实际需要的数据类型。下面通过一个具体示例讲解模板类的生成。

【范例 19-3】类模板的声明和模板类的生成。该范例声明了一个类模板，并根据该类生成了模板类，实现了不同数据类型的出入栈操作，实现代码如代码清单 19-3 所示。

代码清单 19-3

```
1  #include<iostream.h>           //包含头文件
2  const int size=10;             //定义常量
3  template<class T>              //声明类模板
4  class Stack
5  {
6      T stack[size];              //私有成员
7      int t;
8  public:                         //公有成员
9      Stack()                     //定义构造函数
10     {
11         t=0;
12     }
13     push(const T &ch);           //声明成员函数
14     T &pop();
15 };
16 template<class T>
17 Stack<T>::push(const T &ob)      //定义入栈成员函数的功能
18 {
19     if (t==size)                 //栈满
20     {
21         cout<<"stack is full!"<<endl;
22         return 0;
23     }
24     stack[t]=ob;                 //入栈
25     t++;

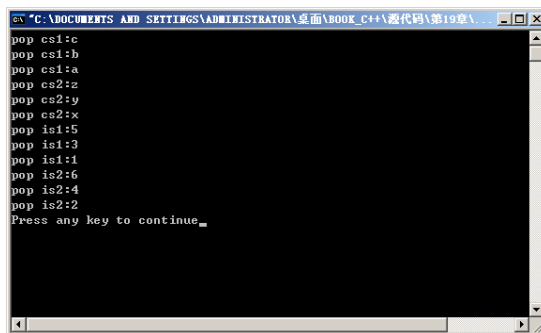
```

```

26 }
27 template<class T>
28 T &Stack<T>::pop() //定义出栈成员函数的功能
29 {
30     if (t==0) //栈空
31     {
32         cout<<"stack is empty!"<<endl;
33         //return 0;
34     }
35     t--; //出栈
36     return stack[t]; //返回栈顶值
37 }
38 void main()
39 {
40     Stack<char>cs1,cs2; //定义对象,生成字符型模板类
41     int i;
42     cs1.push('a'); //字符型数值入栈
43     cs2.push('x');
44     cs1.push('b');
45     cs2.push('y');
46     cs1.push('c');
47     cs2.push('z');
48     for(i=0;i<3;i++) //出栈
49         cout<<"pop cs1:"<<cs1.pop()<<endl;
50     for(i=0;i<3;i++)
51         cout<<"pop cs2:"<<cs2.pop()<<endl;
52     Stack<int>is1,is2; //定义对象,生成整型模板类
53     is1.push(1); //整型数值入栈
54     is2.push(2);
55     is1.push(3);
56     is2.push(4);
57     is1.push(5);
58     is2.push(6);
59     for(i=0;i<3;i++) //出栈
60         cout<<"pop is1:"<<is1.pop()<<endl; //输出出栈结果
61     for(i=0;i<3;i++)
62         cout<<"pop is2:"<<is2.pop()<<endl; //输出出栈结果
63 }

```

【运行结果】同样，在 Visual C++ 中新建一个【C++ Source File】文件，输入上述的代码，编译无误后运行，其结果如图 19-3 所示。



```

C:\DOCUMENTS AND SETTINGS\ADMINISTRATOR\桌面\BOOK_C++\源代码\第19章\
pop cs1:c
pop cs1:b
pop cs1:a
pop cs2:z
pop cs2:y
pop cs2:x
pop is1:5
pop is1:3
pop is1:1
pop is2:6
pop is2:4
pop is2:2
Press any key to continue_

```

图 19-3 类模板的声明和模板类的生成

【范例解析】该范例定义了一个类模板 Stack，在主程序 main() 函数中，分别生成了该模板类的字符型模板类和整型模板类，它们分别可以实现字符型数据出栈和入栈，以及整型数据的出栈和入栈的功能，这就是类模板的定义和模板类的生成。



提示 由于类模板的通用性，在实际程序中，经常使用已经有的类模板生成具体的模板类，如上述代码中的字符型栈和整数型栈都是基于栈类的模板。

19.4 C++标准库概述

一般来说，C++标准可分为两部分：C++语言本身和 C++标准库。由于 ANSI C++的标准未定，因此随着 ANSI C++标准推出的 C++标准库相对于 Visual C++是比较新的。标准库提供了标准的输入/输出、字符串、容器（如矢量、列表和映射等）、非数值运算（如排序、搜索和合并等）和对数值计算的支持。应该说，C/C++包含了相对少的关键字，而且很多最有用的函数都来源于库，C++标准库实现容器和算法的部分就是 STL。

简单地说，C++标准函数库为 C++程序员们提供了一个可扩展的基础性框架，用户从中可以获得极大的便利，同时也可以通过继承现有类，自己编制符合接口规范的容器、算法、迭代器等方式对之进行扩展。简单地说，C++标准库大致包含了如下几个组件。

- C 标准函数库：基本保持了与原有 C 语言程序库的良好兼容，尽管有些微变化。读者需要注意的是，在 C++标准库中存在两套 C 的函数库，一套是带有.h 扩展名的，而另一套则没有，并且它们确实没有太大的不同。
- I/O 流技术。
- String。
- 容器。
- 算法。
- 诊断支持。

其中，标准库中容器和算法这部分，即第 18 章所讲解的标准模板库 STL，事实上，还有第三个构件——迭代器，它让 STL 算法和容器共同工作，如图 19-4 所示。

此外，标准库中东西很多，程序员所选择的类名或函数名很有可能和标准库中的某个名字相同。为了避免这种情况所造成的名字冲突，实际上标准库中的一切被放到了命名空间 std 中，但这带来了新的问题。无数现有 C++代码使用了多年的伪标准库中的功能，如<iostream.h>、<complex.h>、<stdio.h>等头文件功能。

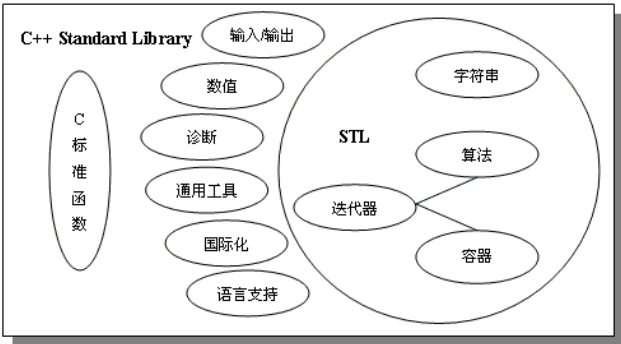


图 19-4 C++标准库的组成

注意 现有软件没有针对使用命名空间而进行设计，如果用 std 来包装标准库导致现有代码不能运行，将会得不偿失。

为减轻程序员负担,标准委员会决定为包装了 `std` 的那部分标准库构件创建新的头文件名。如后来使用的 `<iostream>`、`<cstdio>`、`<complex>` 等都是来自新的命名规则。例如:

- 旧的 C++ 头文件名如 `<iostream.h>` 仍被支持,但其不在命名空间 `std` 中。
- 新的 C++ 头文件如 `<iostream>` 包涵的基本功能和旧的相同,但在 `std` 中。
- 标准 C 头文件如 `<stdio.h>` 继续被支持,不在 `std` 中。
- 具有 C 库功能的新 C++ 头文件具有 `<cstdio>` 这样的名字,和 `<stdio.h>` 相同,在 `std` 中。

19.5 字符串库

前面提到了, C++ 标准库包含内容很多,但由于字符串在程序设计中的重要性,本节将单独对标准库中的字符串库 (`string` 库) 做具体讲解。

19.5.1 读写字符串

读者前面学习了用 `iostream` 标准库来读写内置类型的值,如 `int`、`double` 等数据类型。同样,用户也可以用 `iostream` 和 `string` 标准库使用标准输入/输出操作符来读写 `string` 对象。

【范例 19-4】简单 `string` 的读写,实现了一个简单 `string` 对象的读写操作,实现代码如代码清单 19-4 所示。

代码清单 19-4

```

1  #include <iostream>
2  #include <string>                //包含字符串库
3  using namespace std;
4  void main()
5  {
6      string str;                  //定义字符串对象
7      cout<<"Please input string: "<<endl;
8      cin>>str;                   //接收输入
9      cout<<"The string is : "<<endl;
10     cout<<str<<endl;            //输出
11 }
```

【运行结果】在 Visual C++ 中新建一个 **【C++ Source File】** 文件,输入上述的代码,编译无误后运行,其结果如图 19-5 所示。

【范例解析】读者可以看出,上述代码中定义了一个 `string` 对象 `str`,使用输入/输出流对该对象进行输入/输出操作,但其运行结果并不是所期望的。这是因为从标准输入中读取 `string` 对象,将读入的串存储在 `str` 中,`string` 类型的输入操作符读取并忽略开头所有的空白字符,但在输出时读取字符直至再次遇到空白字符。

输入到程序的是“Hello World!”,而屏幕上将输出“Hello”,空格后面的字符则没有输出。

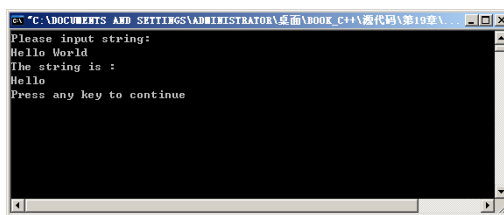


图 19-5 简单 `string` 的读写

19.5.2 字符串赋值

尽管字符串是标准库中的一个类型,但其在总体上设计得和基本数据类型一样方便易用。在标准库中,大多数库类型支持赋值操作,对 `string` 对象来说也是一样的,用户可以把一个 `string` 对象赋值给另一个 `string` 对象,实现简单的字符串赋值操作。



【范例 19-5】字符串赋值。该范例实现了一个简单 string 对象的赋值操作，实现代码如代码清单 19-5 所示。

代码清单 19-5

```

1  #include <iostream>
2  #include <string>                                //包含字符串库
3  using namespace std;
4  void main()
5  {
6      string str1, str2;                            //定义字符串对象
7      cout<<"Please input str1: "<<endl;
8      cin>>str1;
9      str2=str1;                                    //字符串赋值
10     cout<<"str2 is : "<<str2<<endl;              //输出字符串
11 }
```

【运行结果】在 Visual C++ 中新建一个【C++ Source File】文件，输入上述的代码，编译无误后运行，其结果如图 19-6 所示。

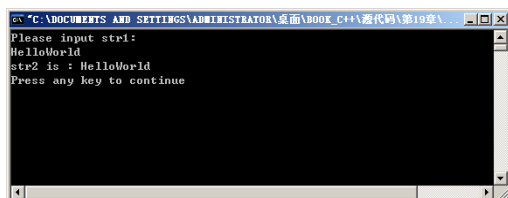


图 19-6 字符串赋值

【范例解析】读者可以看到，赋值操作后，串 str2 就成为包含了 str1 串所有字符的一个副本，即其中存储的数据与 str1 串相同了。

大多数 string 库类型的赋值等操作的实现都会遇到一些效率上的问题，但值得注意的是，从概念上讲，赋值操作是确实需要的工作。它必须先把 st1 占用的相关内存释放掉，然后再分

配给 st1 足够存放 st2 副本的内存空间，最后把 st2 中的所有字符复制到新分配的内存空间。



提示 需要读者注意的是，上述示例中的 str1 仍然不能为中间包含空格的字符串，否则赋值时 str2 将丢弃空格后的字符，造成两个串的值不同。

19.5.3 字符串比较

字符串的比较在实际程序中也是运用较多，因此，string 库定义了几种关系操作符用来比较两个 string 值的大小。一般来说，字符串的比较操作符主要有如下的几种。

- ==操作符：其比较两个 string 对象，如果它们相等，则返回 true。两个 string 对象相等是指它们的长度相同，且含有相同的字符。
- !=操作符：测试两个 string 对象是否不等。
- <操作符：测试一个 string 对象是否小于另一个 string 对象。
- <=操作符：测试一个 string 对象是否小于或等于另一个 string 对象。
- >操作符：测试一个 string 对象是否大于另一个 string 对象。
- >=操作符：测试一个 string 对象是否大于或等于另一个 string 对象。



注意 string 对象比较运算是区分大小写的，即同一个字符的大小写形式被认为是两个不同的字符。在多数计算机上，大写的字母位于小写字母之前，即任何一个大写字母都小于任意的小写字母。

【范例 19-6】字符串比较。该范例实现了两个简单 string 对象的比较操作，实现代码如

代码清单 19-6 所示。

代码清单 19-6

```

1  #include <iostream>
2  #include <string>                //包含字符串库
3  using namespace std;
4  void main()
5  {
6      string str1="hello";          //定义并初始化字符串对象
7      string str2="Hello";
8      string str3="Hello World";
9      int flag;
10     flag=(str1==str2);             //字符串比较
11     if(flag==1)                    //字符串相等
12         cout<<"str1=str2"<<endl;
13     else                            //字符串不等
14     {
15         flag=(str1>str2);          //字符串比较
16         if (flag==1)               //字符串 str1 大于 str2
17             cout<<"str1>str2"<<endl;
18         else                        //字符串 str1 小于 str2
19             cout<<"str1<str2"<<endl;
20     }
21     flag=(str2==str3);             //字符串比较
22     if(flag==1)                    //字符串相等
23         cout<<"str2=str3"<<endl;
24     else                            //字符串不等
25     {
26         flag=(str2>str3);          //字符串比较
27         if (flag==1)               //字符串 str2 大于 str3
28             cout<<"str2>str3"<<endl;
29         else                        //字符串 str2 小于 str3
30             cout<<"str2<str3"<<endl;
31     }
32 }
```

【运行结果】在 Visual C++ 中新建一个【C++ Source File】文件，输入上述的代码，编译无误后运行，其结果如图 19-7 所示。

【范例解析】上述代码中，分别对长度相同但大小写不同的字符串和长度不同的字符串进行了比较，使用多重分支语句判断其之间的关系是等于、大于还是小于，将比较结果输出。

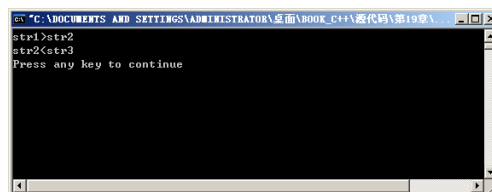


图 19-7 字符串比较

19.5.4 字符串长度和空字符串

在字符串标准库中，经常需要获得字符串的长度，以便进行某种循环。标准库同样提供了求字符串长度的函数：size()。简单地说，string 对象的长度指的是 string 对象中字符的个数，可以通过 size 操作获取。

【范例 19-7】求字符串长度。该范例求出了一个简单 string 对象的长度，实现代码如代码清单 19-7 所示。

代码清单 19-7

```

1  #include <iostream>
2  #include <string>                //包含字符串库
```




```

3  using namespace std;
4  void main()
5  {
6      string str("Welcome to 21 C++");           //定义并初始化字符串
7      int num;
8      num=str.size();                             //求字符串长度
9      cout<<"the length of "<<str<<" is : "<<num<<endl;
                                           //输出长度
10 }

```

【运行结果】在 Visual C++ 中新建一个【C++ Source File】文件，输入上述的代码，编译无误后运行，其结果如图 19-8 所示。

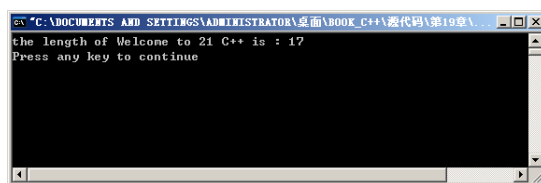


图 19-8 求 string 对象长度

【范例解析】上述代码中，在定义 string 对象 str 的同时就对其进行了初始化赋值，再定义一个整型变量存放其长度，通过 size() 函数获取字符串 str 的长度并输出。



在许多应用程序中，需要判断 string 对象是否为空。这里可以使用 size() 函数求出长度后判断其是否为 0 来实现。

同时，字符串标准库还提供了函数 empty() 来判断一个 string 对象是否为空，该函数返回一个 bool 型值，若为真时表示为空，否则为非空。例如，要判断上述范例 19-7 中的 str 串是否为空，可以将代码 19-7 修改如下。

【范例 19-8】判断 string 对象是否为空。该范例判断了一个 string 对象是否为空，实现代码如代码清单 19-8 所示。

代码清单 19-8

```

1  #include <iostream>
2  #include <string>                               //包含字符串库
3  using namespace std;
4  void main()
5  {
6      string str("Welcome to 21 C++");           //定义并初始化字符串
7      int num;
8      num=str.size();                             //求长度
9      if (num==0)                                 //用 size 操作判断
10         cout<<"The str is empty"<<endl;         //为空
11     else
12         cout<<"The str is not empty"<<endl;     //不为空
13     if(str.empty())                             //用 empty 操作判断
14         cout<<"The str is empty"<<endl;         //为空
15     else
16         cout<<"The str is not empty"<<endl;     //不为空
17 }

```

【运行结果】在 Visual C++ 中新建一个【C++ Source File】文件，输入上述的代码，编译无误后运行，其结果如图 19-9 所示。

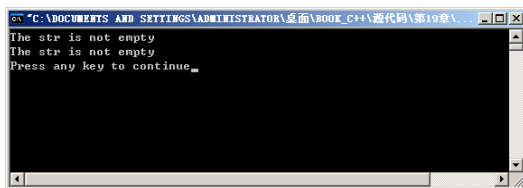


图 19-9 判断字符串是否为空

【代码解析】读者从上述代码中可以看出，它使用了两种方法判断一个 `string` 对象 `str` 是否为空。一种方法是用 `size()` 函数求出 `string` 对象的长度后根据其是否与 0 的比较判断，另一种是直接使用标准库提供的 `empty()` 函数来判断，两种方法实现的结果都是相同的。



注意 一般来说，如果读者并不需要知道 `string` 对象中有多少个字符，只想知道 `size` 是否为 0，可以用 `string` 的成员函数 `empty()` 来实现，这样更为简单。

至此，C++ 的字符串标准库中关于 `string` 对象的常用操作就基本介绍完成了。事实上，标准库中还包含许多其他的函数，可在应用程序中直接调用，读者可以参考相关资料来查阅。

19.6 小结

本章主要介绍了 C++ 中的模板和标准库的相关内容。模板在实际程序中应用较为频繁，其具有宏所不具备的优势，可以提高代码的执行效率。模板分为函数模板和类模板两种，模板都是在实际程序中不能直接调用的，需要先将其生成模板函数和模板类才能使用。C++ 标准库是一个 C++ 函数的仓库，包括前面介绍的 STL。本章主要就标准库中的 `string` 字符串库做了重点介绍，对其中的字符串读写、赋值、比较和判断为空等的函数通过示例进行了讲解。

19.7 习题

1. 编写一个程序，通过函数模板的声明和模板函数的生成，实现不同数据类型数值的交换，如实现整型数据之间的相互交换和浮点型数据之间的相互交换，如图 19-10 所示。

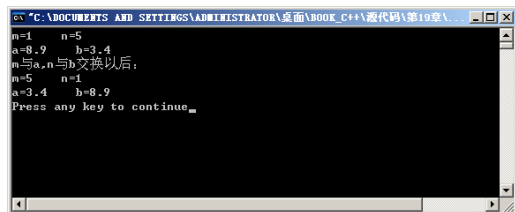


图 19-10 模板应用示例

【解答】该习题主要考查函数模板的相关内容。该程序段要求实现不同数据类型数值的交换，就必须定义函数模板，在模板中完成交换功能，并在主函数中分别对该模板进行整型实例化和浮点型实例化，从而实现交换的目的。其简要的实现代码如下所示。

```
template<typename T>           //声明模板函数，T 为数据类型参数标识符
void swap(T &x, T &y)          //定义模板函数
{
    T z;                       //变量 z 可取任意数据类型及模板参数类型 T
    z=y;                       //交换两个变量的值
    y=x;
    x=z;                       //交换完成
}
```



```

}
void main()                                //主函数
{
    int m=1,n=5;                            //定义整型变量并初始化
    double a=8.9,b=3.4;                    //定义双精度变量并初始化
    cout<<"m="<<m<<"    n="<<n<<endl;    //未交换前输出结果
    cout<<"a="<<a<<"    b="<<b<<endl;    //未交换前输出结果
    swap(m,n);                             //实例化为整型模板函数
    swap(a,b);                             //实例化为双精度型模板函数
    cout<<"m 与 a,n 与 b 交换以后: "<<endl; //输出提示
    cout<<"m="<<m<<"    n="<<n<<endl;    //交换后输出结果
    cout<<"a="<<a<<"    b="<<b<<endl;    //交换后输出结果
}

```

2. 分析以下程序的执行结果。

```

#include<iostream>
template <class T>
class Sample
{
    T n;
public:
    Sample(T i){n=i;}
    void operator++();
    void disp(){cout<<"n="<<n<<endl;}
};
template <class T>
void Sample<T>::operator++()
{
    n+=1;                                // 不能用 n++; 因为 double 型不能用++
}
int main()
{
    Sample<char> s('a');
    s++;
    s.disp();
}

```

【解答】该习题主要考查类模板的使用方法。在上述程序段中，Sample 是一个类模板，由它产生模板类 Sample<char>，通过构造函数给 n 赋初值，通过重载++运算符使 n 增 1，这里 n 由'a'增 1 变成'b'。因此，该习题的输出结果为 n=b。

3. 编写一个使用类模板对数组进行排序、查找和求元素和的程序。

【解答】该习题主要考查类模板的实现问题。在对数组的操作中，排序、查找和求元素和都是使用非常频繁的操作，为了对各种数据类型的数组都能进行上述几种操作，有必要编写一个类模板来实现。该习题可以设计一个类模板 template<class T>class Array，用于对 T 类型的数组进行排序、查找和求元素和，然后由此产生模板类 Array<int>和 Array<double>。其简要的实现代码如下所示。

```

template <class T>
class Array
{
    T *set;
    int n;
public:
    Array(T *data,int i){set=data;n=i;}
    ~Array(){}
    void sort();                // 排序
    int seek(T key);            // 查找指定的元素
}

```

```

    T sum();                // 求和
    void disp();            // 显示所有的元素
};

```

```

int a[]={6,3,8,1,9,4,7,5,2};
double b[]={2.3,6.1,1.5,8.4,6.7,3.8};
Array<int>arr1(a,9);
Array<double>arr2(b,6);

```

4. 分析以下程序的执行结果。

```

#include<iostream>
template<class T>
class Sample
{
    T n;
public:
    Sample(){}
    Sample(T i){n=i;}
    Sample<T>&operator+(const Sample<T>&);
    void disp(){cout<<"n="<<n<<endl;}
};
template<class T>
Sample<T>&Sample<T>::operator+(const Sample<T>&s)
{
    static Sample<T> temp;
    temp.n=n+s.n;
    return temp;
}
int main()
{
    Sample<int>s1(10),s2(20),s3;
    s3=s1+s2;
    s3.disp();
}

```

【解答】该习题主要考查类模板的使用方法和重载运算符的实现。在上述程序段中，Sample 为一个类模板，产生一个模板类 Sample<int>，并建立它的三个对象，调用重载运算符+实现 s1 与 s2 的加法运算，将结果赋给 s3。读者可以看出，重载后的运算符+能够实现两个对象直接的加法运算，因此，该程序段的输出为 s=30。

第 20 章 异常处理

在实际的应用程序设计中，不可避免地会出现程序错误和异常。因此，异常处理是每一种程序设计语言都必须包含的一个部分，C++的异常处理功能非常完善，使用户能够很快发现及捕获异常，尽快地完成程序调试。本章将具体介绍异常处理的基本思想，在 C++中是如何捕获和处理异常，以及 C++中异常的处理机制。

以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为学习的参照。

- 了解错误与异常的概念及其处理基本原则。
- 掌握实际程序中实现异常处理的方法。
- 了解异常处理机制。

20.1 错误与异常

读者可以理解，在实际程序设计中，无论用户的编码技术有多好，出现错误的可能性都很大。因此，程序都必须能处理可能出现的错误和异常。

20.1.1 什么是异常

简单地说，异常就是程序在运行过程中，由于使用环境的变化及用户的操作而产生的错误。例如，内存不足时，应用程序请求分配内存，程序中出现了以零为除数的错误；打印机未打开，导致程序运行中挂接这些设备失败等，都会引发异常。对这些错误，应用程序如果不能进行合适的处理，将会使程序变得非常脆弱，甚至不可使用。

因此，对于这些可以预料的错误，在程序设计时，应编制相应的预防代码或处理代码，以便防止异常发生后造成严重后果。一个应用程序，既要保证其正确性，还应有容错能力，或者说，既要在正确的应用环境中，在用户正确操作时，要运行正常、正确，并且在应用环境出现意外或用户操作不当时，也应有合理的反应。

因此，异常处理对于编写健壮的软件来说无疑是非常重要的，是否有完善的异常处理机制也是评价某一程序设计语言优劣的一个重要标准。

20.1.2 异常处理的基本思想

事实上，所有的程序设计语言对于异常处理的思想基本上都是类似的。简单地说，就是捕获异常后进行处理。

具体地说，处理异常的基本思想是：在底层发生的问题，逐级上报，直到有能力可以处理异常的那级为止。或者说，在应用程序中，如果某个函数发现了错误并引发异常，这个函数就将该异常向上级调用者传递，请求调用者捕获该异常并处理该错误。如果调用者不能处理该错误，就继续向上级调用者传递，直到异常被捕获错误被处理为止。

如果程序最终没有相应的代码处理该异常，那么该异常最后被 C++系统所接受，C++系统就简单地终止程序运行。异常的传递如图 20-1 所示。

由图 20-1 读者可以看出，C++异常处理的目的是在异常发生时，尽可能地减少破坏，使得

其不影响或尽量少地影响程序其他部分的运行。

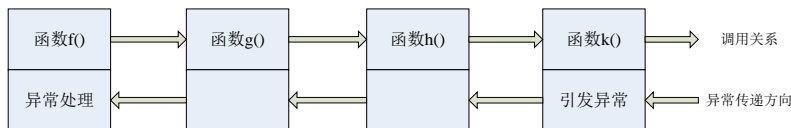


图 20-1 异常的传递方向

提示

总的来说，对于小型程序和大型程序出现异常时，其处理的思想和方法稍有区别。一般来说，当小型程序在出现异常时，则将程序立即中断运行，无条件释放所有资源。

【范例 20-1】异常处理的基本思想。该范例实现当除数为零时，停止运行并给出提示信息，实现代码如代码清单 20-1 所示。

代码清单 20-1

```

1  #include<iostream.h>                                //包含头文件
2  #include<stdlib.h>
3  double fuc(double x, double y)                      //定义函数
4  {
5      if(y==0)                                        //除数为 0
6      {
7          cerr<<"error of dividing zero.\n";        //输出错误信息
8          exit(1);                                    //异常退出程序
9      }
10     return x/y;                                     //返回两个数的商
11 }
12 void main()
13 {
14     double res;
15     res=fuc(2,3);                                    //调用函数
16     cout<<"The result of x/y is : "<<res<<endl;    //输出正确结果
17     res=fuc(4,0);                                    //除数为 0 发生异常
18     cout<<"The result of x/y is : "<<res<<endl;    //不执行该语句
19 }
```

【运行结果】同样的，在 Visual C++中新建一个【C++ Source File】文件，输入上述的代码，编译无误后运行，其结果如图 20-2 所示。

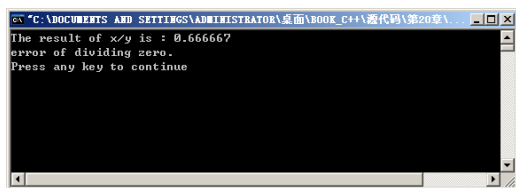


图 20-2 异常处理

【范例解析】上述代码中，定义了函数 fuc()，用于返回两个数的商，并处理当除数为 0 时的异常，在主函数 main()中第 17 行调用 fuc()函数时出现了一个除数为 0 的异常，其并不输出两个数的商，而是调用代码中第 4~9 行的异常处理，输出错误信息并异常退出程序。

注意

exit()函数用于退出程序，该函数可加参数，其中 exit(0)表示正常退出程序，而 exit(1)表示异常退出程序。



当然，如果是在大中型程序中，上述处理方法就过于简单了。这是因为在大中型程序中，函数之间有着明确的分工和复杂的调用关系。由于在这些程序中，发现错误的程序往往在函数调用链的底层，如果简单地在发现错误的函数中处理异常，就没有机会把上层函数已经完成的一些工作做妥善的处理。

20.2 异常处理的实现

一般来说，在 C++ 中对异常处理的实现分为捕获异常和处理异常两个步骤，本节将讲解 C++ 中对这两个步骤的实现。

20.2.1 使用 try/catch 捕获异常

通过前面的介绍，读者已经了解到，对异常的处理之前首先需要捕获到异常。在 C++ 中，提供了语句 try/catch 来捕获异常，其中，try 和 catch 分别用于定义异常和定义异常处理。定义异常是将可能产生错误的语句放在 try 语句块中。其格式是：

```
try
{
    可能产生错误的语句
}
```

定义异常处理是将异常处理的语句放在 catch 语句块中，以便异常被传递来时处理。通常，异常处理是放在 try 语句块后的由若干个 catch 语句组成的程序，其格式是：

```
catch(异常类型声明 1)
{
    异常处理语句块 1
}
catch(异常类型声明 2)
{
    异常处理语句块 2
}
.....
catch(异常类型声明 n)
{
    异常处理语句块 n
}
```

例如，下列语句使用 try/catch 捕获异常，并定义捕获后对异常的处理。

```
try
{
    string str = null;           //定义字符串对象
    ProcessString(str);         //执行某个函数
}
catch (Exception e)             //定义对异常的处理
{
    cout<<"Process is error";
    exit(1);                     //异常退出程序
}
```



提示 在使用 catch 语句定义对异常的处理时，其中的参数可以只为某个数据类型，如 catch(int) 的形式，在具体的环境中其类型不同。

20.2.2 使用 throw 抛出异常

抛出异常（也称为抛出异常）即检测是否产生异常，在 C++ 中，其采用 `throw` 语句来实现，如果检测到产生异常，则抛出异常。该语句的格式为：

`throw 表达式;`

如果在 `try` 语句块的程序段中（包括在其中调用的函数）发现了异常，且抛弃了该异常，则这个异常就可以被 `try` 语句块后的某个 `catch` 语句所捕获并处理，捕获和处理的条件是被抛弃的异常的类型与 `catch` 语句的异常类型相匹配。由于 C++ 使用数据类型来区分不同的异常，因此在判断异常时，`throw` 语句中的表达式的值就没有实际意义，而表达式的类型就特别重要。

【范例 20-2】 处理除数为 0 的异常。该范例将上述除数为 0 的异常可以用 `try/catch` 语句来捕获异常，并使用 `throw` 语句来抛出异常，从而实现异常处理，实现代码如代码清单 20-2 所示。

代码清单 20-2

```
1  #include<iostream.h>                                //包含头文件
2  #include<stdlib.h>
3  double fuc(double x, double y)                      //定义函数
4  {
5      if(y==0)
6      {
7          throw y;                                    //除数为 0，抛出异常
8      }
9      return x/y;                                     //否则返回两个数的商
10 }
11 void main()
12 {
13     double res;
14     try                                              //定义异常
15     {
16         res=fuc(2,3);
17         cout<<"The result of x/y is : "<<res<<endl;
18         res=fuc(4,0);                               //出现异常
19     }
20     catch(double)                                  //捕获并处理异常
21     {
22         cerr<<"error of dividing zero.\n";
23         exit(1);                                    //异常退出程序
24     }
25 }
```

【运行结果】 在 Visual C++ 中新建一个【C++ Source File】文件，输入上述的代码，编译无误后运行，其结果如图 20-3 所示。

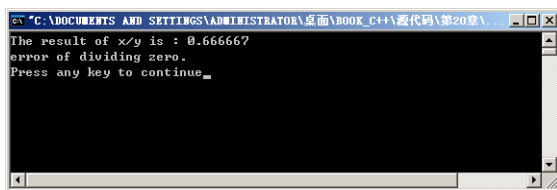


图 20-3 处理除数为 0 的异常

【范例解析】 上述代码中，在主函数 `main()` 的第 14~19 行中使用了 `try` 语句定义异常，其中包含 3 条有可能出现异常的语句，它们为调用两个数相除的函数。在代码的第 20~24 行定义了异常处理，即捕获异常后执行该段代码中的语句。此外，在函数 `fuc()` 的代码 5~8 行通过



throw 语句抛出异常。



一般来说, throw 语句通常与 try- catch 或 try-finally 语句一起使用, 可以使用 throw 语句显式引发异常。

20.2.3 应用示例

为了让读者更好地理解异常是如何被捕获及如何被处理的, 本节给出一个应用示例及其执行过程分析。

【范例 20-3】异常处理应用实例。该范例给出使用 try/catch 语句实现捕获和处理异常, 实现代码如代码清单 20-3 所示。

代码清单 20-3

```
1  #include<fstream.h>                                //包含头文件
2  double Div(double, double);                          //声明函数
3  void main()
4  {
5      try                                              //定义异常
6      {
7          cout<<"7.3/2.0="<<Div(7.3,2.0)<<endl;
8          cout<<"7.3/0.0="<<Div(7.3,0.0)<<endl;      //出现异常
9          cout<<"7.3/1.0="<<Div(7.3,1.0)<<endl;
10     }
11     catch(double)                                    //定义异常处理
12     {
13         cout<<"exception of deviding zero.\n";      //输出错误信息
14     }
15     cout<<"that is ok.\n";
16 }
17 double Div(double a, double b)                      //定义函数
18 {
19     if (b==0.0)
20         throw b;                                    //抛出异常
21     return a/b;
22 }
```

【运行结果】在 Visual C++中新建一个【C++ Source File】文件, 输入上述的代码, 编译无误后运行, 其结果如图 20-4 所示。

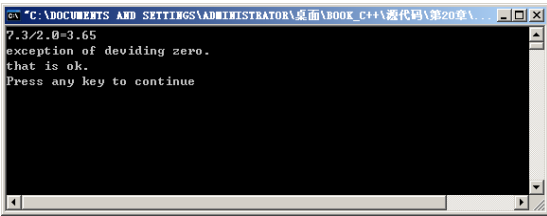


图 20-4 异常处理应用示例

【范例解析】上述代码中, 第 8 行中调用函数 Div(7.3,0.0)时, 控制转移到函数 Div(7.3,2.0)内执行, 这时 b==0.0 为真, 发生异常, 函数 Div()被退栈处理, 紧跟调用函数 Div()后面的语句:

cout<<"7.3/1.0="<<Div(7.3,1.0)<<endl;

不再被执行, 而异常被

```
catch(double)
{
    cout<<"exception of deviding zero.\n";
}
```

捕获, 执行完异常处理, 程序紧接着执行异常处理后面的语句:

```
cout<<"that is ok.\n";
```

其执行流程如图 20-5 所示。

从上述示例读者可以看出, 异常处理的执行过程是:

- ① 控制通过正常的顺序执行到 try 语句, 然后执行 try 语句块内的程序段。
- ② 如果在 try 语句块执行期间没有发生异常, 则 catch 语句块不被执行。
- ③ 如果在 try 语句块执行期间或在该语句块直接或间接调用的任何函数中发生了异常, 并将异常抛弃, 则该异常将沿调用链上传, 直到找到与该异常类型相匹配的 catch 语句块来处理异常为止。异常处理后, 执行所有 catch 语句块的后续程序。
- ④ 如果未找到与该异常类型相匹配的 catch 语句块, 则由 C++ 终止程序的运行。

此外, 读者在使用异常处理时, 要注意以下问题:

- C++ 只处理放在 try 语句块内受监控的过程的异常, 那些不受监控的过程的异常, C++ 是不会处理的。
- 在 try 语句块之后必须紧跟一个或多个 catch 语句块, 以便对发生的异常进行处理。在 try 语句块出现之前, 不能出现 catch 语句块。
- catch 语句的括号中只能有一个形参, 但该形参是可选的, 而形参的数据类型不能默认, 必须保留, 因为捕获是利用数据类型的匹配实现的。
- 抛出异常与处理异常可以放在不同的函数中。
- catch(…)语句可以捕获全部异常, 因此, 若使用这个语句, 应将它放置在所有的 catch 语句之后。

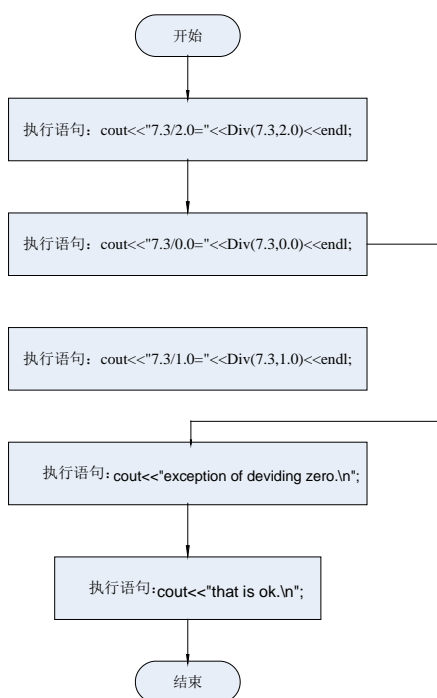


图 20-5 异常处理执行流程



除此之外, 为了增强程序的可读性, C++ 允许在函数的声明中注明函数可能抛弃的异常类型, 其语法为:

```
返回值类型 函数名(形参列表) throw(异常类型 1, 异常类型 2, ...)
```

例如, 上述范例中的 Div 函数的声明可写为:

```
double Div(double x, double y) throw(int);
```

20.3 类和结构的异常处理

C++ 异常处理的真正能力不仅在于它能处理各种不同类型的异常, 还在于它具有在异常抛弃前为构造的所有局部对象自动调用析构函数的能力。



20.3.1 异常处理中的构造和析构

当在程序中找到一个匹配的 `catch` 异常处理后，如果 `catch()` 语句的异常类型声明是一个值参数，则其初始化方式是复制被抛弃的异常对象；如果 `catch()` 语句的异常类型声明是一个引用，则其初始化方式是使该引用指向异常对象。

【范例 20-4】使用带析构的类的异常处理。该范例包含类及其构造函数与析构函数，其进行异常处理时析构函数的调用会有所不同，代码如代码清单 20-4 所示。

代码清单 20-4

```
1  #include<iostream.h>
2  class expt                                //定义类 expt
3  {
4  public:                                    //定义公有成员
5      expt()                                //定义构造函数
6      {
7          cout<<"structor of expt"<<endl;
8      }
9      ~ expt()                               //定义析构函数
10     {
11         cout<<"destructor of expt"<<endl;
12     }
13 };
14 class demo                                //定义类 demo
15 {
16 public:
17     demo()                                  //定义构造函数
18     {
19         cout<<"structor of demo"<<endl;
20     }
21     ~demo()                                 //定义析构函数
22     {
23         cout<<"destructor of demo"<<endl;
24     }
25 };
26 void fuc1()                                //定义函数
27 {
28     int s=0;
29     demo d;                                //声明 demo 类的对象
30     throw s;                               //抛出异常
31 }
32 void fuc2()
33 {
34     expt e;                                //声明 expt 类的对象
35     fuc1();                                //调用函数 fuc1
36 }
37 void main()
38 {
39     try                                     //定义异常
40     {
41         fuc2();                             //调用函数
42     }
43     catch(int)                             //定义异常处理
44     {
45         cout<<"catch int exception"<<endl;
46     }
47     cout<<"continue main()"<<endl;
48 }
```

【运行结果】在 Visual C++ 中新建一个【C++ Source File】文件，输入上述的代码，编译无误后运行，其结果如图 20-6 所示。

【范例解析】上述代码中定义了两个类 `expt` 和 `demo`，在函数 `fuc2()` 中创建了 `expt` 的对象 `e`，在函数 `fuc1()` 中创建了 `demo` 的对象 `d`，并抛出异常。在主函数 `main()` 中，使用 `try/catch` 语句捕获并处理异常。从运行结果可以看出，在抛出异常前，创建了两个对象 `e` 和 `d`，在抛出异常后，这两个对象被按与创建的相反顺序调用析构函数销毁。

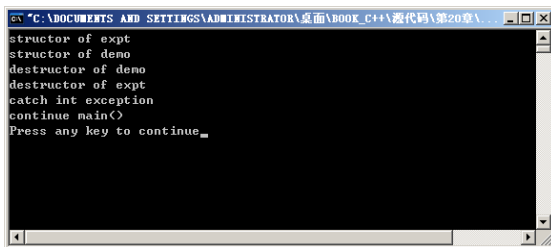


图 20-6 使用带析构的类的异常处理



注意

当 `catch()` 语句的异常类型参数被初始化后，便开始了栈的展开过程，包括从对应的 `try` 语句块开始到异常被抛弃之间对构造的所有自动对象进行析构。析构的顺序与构造的顺序相反。然后程序从最后一个 `catch` 处理之后开始恢复。

20.3.2 处理结构类型的异常

由于类与结构有相似性，对于结构类型的异常处理也有些需要读者注意的地方。主要在于 `throw` 语句中的表达式类型，一般使用结构类型。

【范例 20-5】处理结构类型的异常。该范例包含结构类型进行异常处理时，需要注意一些事项，代码如代码清单 20-5 所示。

代码清单 20-5

```

1  #include<iostream.h>
2  struct aircraft                                //定义结构体
3  {
4      char *aircraftttype;                        //定义结构体成员
5      float len;
6  };
7  void myfuc()                                    //定义函数
8  {
9      aircraft a;                                //定义结构体变量
10     a.aircraftttype="queen";                    //结构体变量初始化
11     a.len=125;
12     throw a;                                    //抛出异常
13 }
14 void main()
15 {
16     try                                          //定义异常
17     {
18         myfuc();                                //调用函数
19     }
20     catch(aircraft b)                          //定义异常的处理,
                                                //参数为结构体类型
21     {

```



```

22         cout<<"aircraft type is:"<<b. aircrafttype<<"\n"; //输出
23     }
24 }

```

【运行结果】在 Visual C++ 中新建一个【C++ Source File】文件，输入上述的代码，编译无误后运行，其结果如图 20-7 所示。

【范例解析】读者可以看出，该范例与其他范例不同点在于其是结构体数据类型。在上述代码的第 20 行使用 catch 语句定义异常处理语句时，其参数为结构类型。

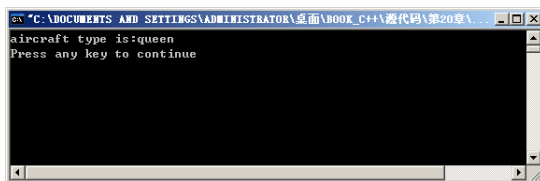


图 20-7 处理结构类型的异常

20.4 异常处理机制

C++ 并不是第一个对结构化运行期错误处理进行支持的语言。早在 20 世纪 60 年代，PL/1 语言就提供了一种内建的异常处理机制；Ada 语言也在 20 世纪 80 年代提供了自己的异常处理，而 C++ 是在 1989 年时才有了异常处理机制。但是，C++ 的异常处理是独一无二的，并且其已经作为一种模型出现在一些新产生的语言之中。



提示 C++ 异常处理机制是一个用来有效地处理运行错误的非常强大且灵活的工具，它提供了更多的弹性、安全性和稳固性，克服了传统方法所带来的问题。

事实上，C++ 中的异常处理机制是一种把控制权从异常发生的地点转移到一个匹配的处理函数或功能块的机制。其中，异常可以是内建数据类型变量，也可以是对象。一般来说，异常处理机制包括 4 个部分。

- try 语句块：即一个定义异常的语句块。
- catch 语句块：即一个或多个和 try 语句块相关的处理，它们放在 catch 语句块中。
- throw 表达式：即抛出异常语句。
- 异常本身。

一般来说，try 语句块包含可能抛出异常的代码。例如，下列语句可能引发内存空间溢出的异常，其就包含在 try 语句中。

```

try
{
    int * p = new int[1000000];
}

```

一个 try 语句块后面将跟有一个或多个 catch 语句，其中，每一个 catch 语句可以处理不同类型的异常。例如：

```

try
{
    int * p = new int[1000000];
    //...
}
catch(std::bad_alloc & )           //内存空间不够，分配内存失败
{
}

```

```
catch (std::bad_cast&)
{
}
```

//转型失败, 分配内存失败

catch 语句块仅仅被在 try 语句块中的 throw 表达式及函数所调用。其中, throw 表达式包括一个关键字 throw 及相关参数。例如:

```
try
{
    throw 5;
}
catch(int n)
{
}
```

读者需要注意的是, throw 表达式和返回语句很相似。此外, throw 语句可以没有操作数, 其格式如下所示:

```
throw;
```



注意 一般来说, 如果目前没有异常被处理, 那么执行一个没有操作数的 throw 语句后, 编译系统将会调用 terminate() 函数结束程序。

当一个异常被抛出后, C++ 运行机制首先在当前的作用域寻找合适的处理 catch 语句块。如果不存在这样一个处理, 那么将会离开当前的作用域, 进入更外围的一层继续寻找。这个过程不断地进行下去直到合适的处理被找到为止。此时堆栈已经被解开, 并且所有的局部对象被销毁。如果始终都没有找到合适的处理, 那么程序将会终止。

【范例 20-6】求一元二次方程的实根, 要求加上异常处理, 判断 $b^2-4*a*c$ 是否大于 0, 成立则求两个实根, 否则要求重新输入, 实现代码如代码清单 20-6 所示。

代码清单 20-6

```
1  #include <iostream>
2  #include <math.h>                                //包含头文件
3  using namespace std;                             //使用命名空间
4  double sqrt_delta(double d)                       //定义函数
5  {
6      if(d < 0)
7          throw 1;                                  //抛出异常
8      return sqrt(d);                               //返回平方根值
9  }
10 double delta(double a, double b, double c)        //定义函数
11 {
12     double d = b * b - 4 * a * c;
13     return sqrt_delta(d);                          //调用 sqrt_delta() 函数
14 }
15 void main()
16 {
17     double a, b, c;
18     cout << "please input a, b, c" << endl;
19     cin >> a >> b >> c;                            //接收输入
20     while(true)                                    //循环
21     {
22         try                                         //定义异常
23         {
24             double d = delta(a, b, c);            //调用函数
25             cout << "x1: " << (d - b) / (2 * a);
26             cout << endl;
```



```

27         cout << "x2: " << -(b + d) / (2 * a);
28         cout << endl;
29         break;                                //跳出循环
30     }
31     catch(int)                                //定义异常处理
32     {
33         cout << "delta < 0, please reenter a, b, c.";
34                                     //重新输入系数
35         cin >> a >> b >> c;
36     }
37 }

```

【运行结果】在 Visual C++ 中新建一个【C++ Source File】文件，输入上述的代码，编译无误后运行，其结果如图 20-8 所示。

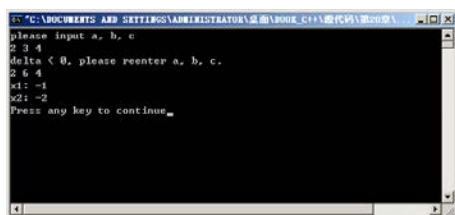


图 20-8 求一元二次方程的实根

【范例解析】上述范例实现了对于用户输入的一元二次方程系数的判断，由于只有方程的系数符合 $b^2 - 4ac > 0$ 的条件时才有实根，所以上述代码中的 `sqrt_delta()` 函数中包含了异常定义。在上述代码的第 31~35 行中的 `catch` 语句块中包含了对该异常的处理。



提示 `catch` 语句后的参数（数据类型）需要与 `throw` 语句后的表达式数据类型相同。如上述代码中第 7 行 `throw` 语句后的表达式为 1，则 `catch` 语句后的参数为 `int`。

20.5 小结

本章主要介绍了 C++ 中关于异常处理及其机制的内容。异常处理是所有程序设计语言都需要包含的一个部分，C++ 的异常处理机制主要由定义异常、定义异常处理和 `throw` 语句等组成。对于 C++ 中处理异常的语句主要包括：`try` 语句、`catch` 语句和 `throw` 语句等。对于不同类型的异常处理，其 `throw` 语句后的表达式类型较为重要，尤其是当处理类型为结构体时。

20.6 习题

1. 打开文件发生错误是经常发生的情况。设计一个程序，该程序可以处理一个处理文件时发生的异常。

【解答】该习题主要考查 C++ 异常处理机制的应用。打开文件需要使用到文件类并创建输入/输出流对象，通过调用对象的 `fail()` 成员函数判断其是否打开成功。此处需要使用到 `try/catch` 语句来进行异常的定义和捕获，其中异常定义语句写在 `try` 语句中，异常处理语句写在 `catch` 语句中。其简要的实现代码如下所示。

```

#include <fstream.h>                                //包含文件打开头文件
#include <iostream.h>                                //包含输入/输出头文件
#include <stdlib.h>                                    //包含库函数头文件
void main(int argc, char *argv)                      //带参数的main()函数
{
    ifstream source(argv[1]);                        //打开文件
    char line[128];                                  //定义字符数组
    try                                              //定义异常
    {
        if (source.fail())                            //打开失败

```

```

        throw argv[1]; //抛出异常
    }
    catch(char * s) //定义异常处理
    {
        cout<<"error opening the file"<<s<<endl; //输出错误信息
        exit(1); //异常退出
    }
    while(!source.eof()) //文件未结束
    {
        source.getline(line, sizeof(line)); //取出文件中内容
        cout<<line<<endl; //输出
    }
    source.close(); //关闭对象
}

```

2. 定义一个异常处理类, 该类需能捕获错误类型并返回, 在主函数中当用户输入整数 1 时发生异常, 并调用类的成员函数进行异常处理。

【解答】该习题主要考查异常处理的实现。首先在类中定义一个错误类型作为私有成员, 而成员函数则是返回错误类型并对错误进行处理。在主函数中接收用户输入, 当输入为整数 1 时通过 try...catch 语句捕获该异常并进行处理。其简要的实现代码如下所示。

```

class myError
{
private:
    string sz_Error;
public:
    myError(string sz) {
        this->sz_Error = sz; }
    string what() {
        return sz_Error; }
};

int main()
{
    int i;
    cin>>i;
    try
    {
        if (i == 1) throw myError("error");
    }
    catch (myError e)
    {
        cout < < "Error:" < < e.what() < < endl;
    }
}

```

3. 写出下列程序的运行结果。

```

#include <iostream.>
int Div(int x,int y)
{
    if(y==0)
        throw y;
    return x/y;
}
int main()
{
    try
    {
        cout<<"7/3="<<Div(7,3)<<endl;
        cout<<"9/0="<<Div(9,0)<<endl;
    }
}

```




```

        cout<<"8/4="<<Div(8,4)<<endl;
    }
    catch(int)
    {
        cout<<"Exception of dividing zero."<<endl;
    }
    cout<<"It is OK."<<endl;
}

```

【解答】该习题主要考查具体程序中如何进行异常捕获并处理。上述程序中，定义了函数 Div，其实现两个整数的除法，当除数为 0 时抛出异常。在主函数中使用了语句 try...catch 进行异常处理，分别运行 7/3、9/0 和 8/4 等运算，如出现异常则进行异常处理。根据 try...catch 语句的执行流程，在处理 9/0 表达式时出现异常，则程序跳过后面的语句，直接执行 catch 中的语句。因此，该程序的输出如下所示。

```

7/3=2
Exception of dividing zero.
It is OK.

```

4. 请写出运行下列程序的结果。

```

void testfun(int test)
{
    try
    {
        if(test)
            throw test;
        else
            throw "it is a zero";
    }
    catch(int i)
    {
        cout<<"Except occurred: "<<i<<endl;
    }
    catch(const char *s)
    {
        cout<<"Except occurred: "<<s<<endl;
    }
}

int main()
{
    testfun(10);
    testfun(100);
    testfun(0);
}

```

【分析】该习题主要考查对异常处理过程的理解。主函数第一次调用 testfun 函数时传递参数 10，10 不等于 0，所以执行抛出异常语句 throw 10；10 为 int 型，所以被 catch(int i) 捕获，所以输出 Except occurred:10。同理第二次调用 testfun 函数时输出 Except occurred:100。第三次调用 testfun 函数时参数为 0，所以执行抛出异常语句 throw "it is a zero";该异常被 catch(const char *s) 捕获，输出 Except occurred: it is a zero。因此上述程序段的运行结果为：

```

Except occurred:10
Except occurred:100
Except occurred: it is a zero

```

第六篇 实例篇

第 21 章 简单学生成绩管理系统 开发实例

通过前面章节的学习，读者对于 C++ 语言的基本概念和重要知识点应有一个基本的理解。前面章节系统地介绍了 C++ 的基本内容，并通过大量的范例来实现演示其使用方法和特性。为了使读者能够全面地掌握使用 C++ 进行应用系统开发的基本步骤，本章以学生成绩管理系统为例，系统地介绍通过 C++ 进行应用程序开发的流程。

以下是对读者在学习本章内容时所提出的几个基本要求，也是本章希望能够达到的目的，让读者在学习本章内容时可以作为学习的参照。

- 了解开发一个应用程序的软件工程生命周期。
- 掌握使用 C++ 开发具体应用程序的流程。
- 掌握使用 Visual C++ 6.0 的控制台程序开发 C++ 应用程序。

21.1 需求分析

一般来说，对于一个简单的学生成绩管理系统，主要包括成绩录入、计算总分和平均分、成绩排名和成绩查询几方面的功能。事实上，一个完整的成绩管理系统应该是一个数据库应用程序，它必须拥有数据库管理和用户管理等功能，为简单起见并突出 C++ 的重点，本章介绍的简单学生成绩管理系统不考虑这些功能。

此外，由于本章前面章节使用的 C++ 编译环境都是 Visual C++ 6.0，此处也同样采用该环境。但是，为突出 C++ 的特性，该管理系统不采用 MFC 编写成 Windows 下的应用程序，而只采用 Visual C++ 6.0 提供的控制台程序来实现。

根据如上的分析，下面给出简单学生成绩管理系统的主要实现功能：

- 提供成绩录入界面。
- 统计每个学生的总分和平均分。
- 按总分由大到小排出名次。
- 提供成绩查询功能，即任意输入一个学号，能够查找出该学生在班级中的排名及其考试成绩。

为简单起见，该范例中的学生只有三门课程的成绩：语文、数学、英语，需要输入的学生信息也只有学生的学号和姓名这两个字段。



提示 在实际的应用中，需求分析要结合现有的资源和客户的需求，以便根据需求分析的结果设计出合理的系统结构。

21.2 总体设计

总体设计阶段即系统的概要设计，需要完成对系统结构的分析和设计，以及设计系统需要的主要数据结构。本节将基于需求分析的结果，给出简单学生成绩管理系统的总体结构。



根据需求分析的结果，本系统至少要分为以下几个模块：main 函数模块、成绩录入模块、成绩统计模块、成绩排名模块和成绩查询模块。其中，各模块的功能说明如下：

- main 函数模块的主要功能为提供程序入口、设置前期环境、调用主要的执行函数和程序结束前的数据处理。
- 成绩录入模块的主要功能为提供简单友好的成绩录入界面，将输入的成绩存储在对应的数据结构中。
- 成绩统计模块的主要功能为统计每个学生的总分和平均分并输出。
- 成绩排名模块的主要功能为按总分由大到小排出名次并输出。
- 成绩查询模块的主要功能为根据用户输入一个学号，能够查找出该学生在班级中的排名及其考试成绩。

根据上述描述，给出该系统的总体设计图，如图 21-1 所示。

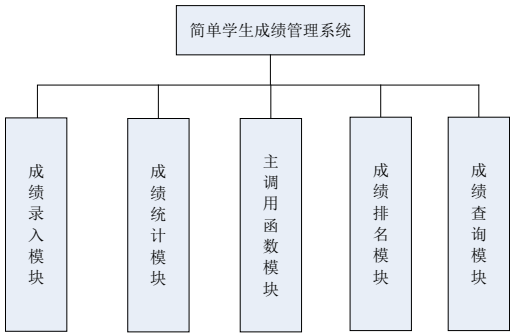


图 21-1 总体设计

此外，由于该系统涉及多个学生的成绩存储，而为简单起见这里没有设置数据库。因此，就应该定义一个数据结构，用于存储这些信息。

通过前面章节的学习，读者知道，结构体和类都可以实现存储多个不同数据类型数据的存储。为体现 C++ 的特性，采用类来存储这些数据类型，该类声明如下：

```
class Student
{
public:
    char number[Max];           //存储学生学号
    char name[Max];             //存储学生姓名
    double chinese;              //存储语文成绩
    double math;                 //存储数学成绩
    double english;             //存储英语成绩
    double total;                //总成绩
    double average;             //平均成绩
    int rank;                    //排名
};
```

注意 该系统中所有的数据都以上述方式存储在计算机中，为简单起见，此处没有就数据成员的安全性考虑，因此数据成员均为公有成员。

21.3 功能模块实现

下面将依次介绍成绩录入模块、成绩统计模块、成绩排名模块、成绩查询等模块和主函数 main 模块的实现。

21.3.1 成绩录入模块

成绩录入模块是简单成绩管理系统首先要执行的一个模块,只有当数据结构中存储有成绩等数据后才能进行其他的诸如统计、排名和查询等功能。

简单地说,成绩录入就是对前面定义的类 `Student` 中的成员进行赋值。其中,用户需要输入的是学生的学号、姓名两个基本信息和语文、数学和英语三门成绩,其余总成绩、平均成绩和排名由后续的计算函数来实现。实现成绩录入模块的函数代码如代码清单 21-1 所示。

代码清单 21-1

```

1 void setData(Student &s)
2 {
3     cout<<"输入学号,姓名,语文,数学,英语成绩:"; //录入数据
4     cin>>s.number>>s.name>>s.chinese>>s.math>>s.english; //接收输入并存储到成员变量中
5     s.total=0; //初始化变量
6     s.average=0; //初始化变量
7     s.rank=0; //初始化变量
8 }
```

上述代码中,第 1 行代码中,函数 `setData()` 为成绩录入的函数名称,其参数为类 `Student` 的一个对象。第 3~4 行接收用户从键盘输入的信息并存储在该对象的成员中,第 5~7 行对该对象的其他成员初始化。该函数运行时其结果如图 21-2 所示。

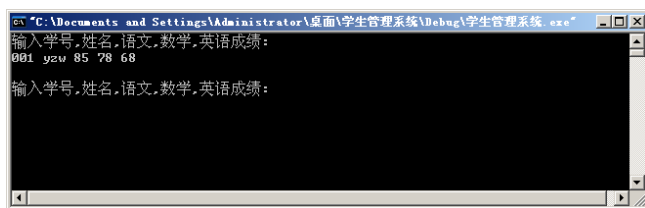


图 21-2 成绩录入界面

读者可以看出,在成绩录入中,只需输入学生的学号、姓名、语文、数学和英语 5 个值即可,其他总分和平均分等值由程序自动赋初值。



提示 在学生成绩管理系统中,往往需要录入的并不是一个同学的成绩,而是多条记录,如 10 位同学就需要录入 10 次,这可以在主程序中通过循环重复调用该函数。

例如,需要输入 10 位同学的成绩,在主函数中调用时代码如下:

```

for(int i=0;i<10;i++) //循环输入
{
    cout<<"下面输入第"<<i+1<<"位同学的数据:"<<endl; //提示
    setData(S[i]); //调用成绩录入函数
    cout<<endl; //换行
}
```

21.3.2 成绩统计模块

成绩统计模块需要建立在成绩录入模块的基础之上,当系统中已经包含一些数据记录后才能对这些成绩进行求总分、平均分等操作。本模块包含对输入的数据计算总分和计算平均分,其中, `cout()` 函数用于计算一个同学的总分和平均分,而 `getAverage()` 函数则求出整个班级所有的三门课程的平均成绩,其函数代码如代码清单 21-2 所示。



代码清单 21-2

```
1 void count(Student &s) //成绩统计函数
2 {
3     s.total=s.chinese+s.math+s.english; //求三门课程的总分
4     s.average=s.total/3; //求三门课程的平均分
5 }
6 double getAverage(Student S[],int N) //求平均分函数
7 {
8     double Average=0; //变量定义并初始化
9     for(int i=0;i<N;i++) //循环
10         Average+=(S[i].chinese+S[i].math+S[i].english); //求所有学生的所有成绩和
11     Average/=(N*3); //求所有学生的成绩平均值
12     return Average; //返回该平均值
13 }
```

其中，计算整个班级所有的三门课程的成绩 getAverage()函数的实现流程如图 21-3 所示。

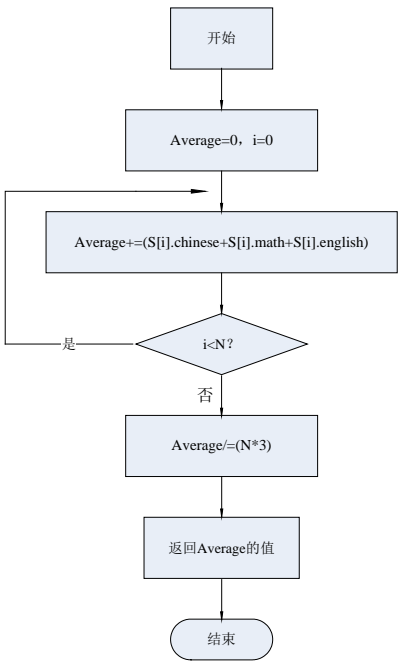


图 21-3 统计平均分流程

注意 该统计模块中，需要统计一个班级同学的总分和平均分，就需要确定班上同学的人数，这个人数就放在常量 N 中。

21.3.3 成绩排名模块

在获得整个班级的总分和平均分之后，就可以对该班级中的同学按照其某个参数进行排名了。在该系统中，使用每位同学的平均成绩对其进行排名。前面内容提到过，排序算法有许多种，如冒泡排序、选择排序等，此处使用插入排序来实现。

插入排序的基本思想是：每次将一个待排序的记录，按其关键字大小插入到前面已经排好序的子序列中的适当位置，直到全部记录插入完成为止，其算法流程如图 21-4 所示。

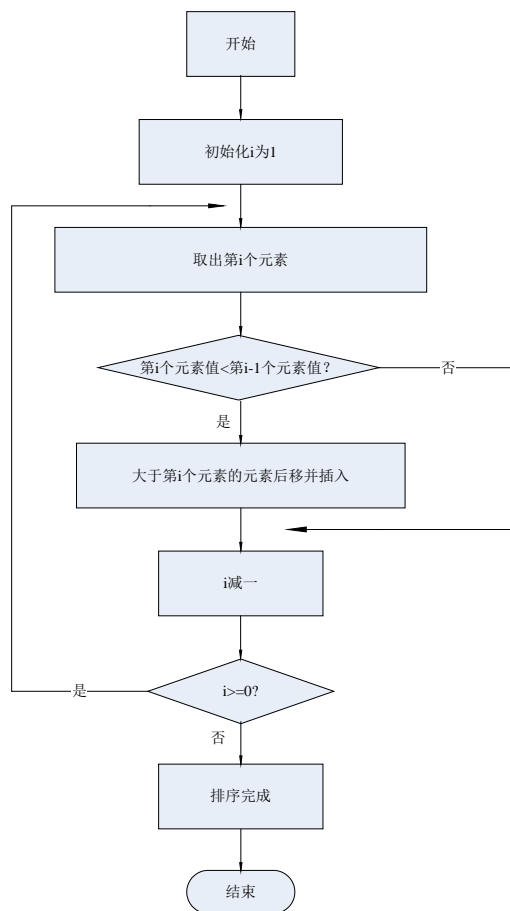


图 21-4 插入排序算法流程图

根据上述的算法流程图，结合 C++ 的语法格式，那么成绩排名模块的实现代码如代码清单 21-3 所示。

代码清单 21-3

```

1  void sort(Student S[],int N)                //插入法排序
2  {
3      int index;                               //定义变量
4      Student inserter;                       //创建对象
5      for(int i=1;i<N;i++)                     //循环
6      {
7          inserter=S[i];                      //对象初始化
8          index=i-1;                          //变量初始化
9          while(index>=0&&inserter.average>S[index].average) //比较
10         {
11             S[index+1]=S[index];             //元素后移
12             index--;
13         }
14         S[index+1]=inserter;                 //插入该元素
15     }
16     for(int j=0;j<N;j++)
17         S[j].rank=j+1;                      //设置排名
18 }

```



上述代码中，前面第 1~15 行代码都是实现按照学生的平均成绩进行从大到小的排序，第 16~17 行代码将排序的名次写入到对象的成员中。

21.3.4 成绩查询模块

该系统中的成绩查询功能是根据用户输入的学生的学号作为关键字，在数据结构中进行查询，并将查询结果即该记录的位置返回，其实现函数如代码清单 21-4 所示。

代码清单 21-4

```
1  int search(Student S[],int N,char *n)           //成绩查询函数
2  {
3      for(int i=0;i<N;i++)                         //在所有记录中查询
4      {
5          if(strcmp(S[i].number,n)==0)             //找到
6              return i;                             //返回结果所在的位置
7      }
8      return -1;                                    //没有找到
9  }
```

上述代码中，第 1 行代码函数的声明中，参数字符串 n 即为用户输入的需要查询的学生学号。将该学号与所有数据元素进行比较，找到则返回该元素的位置，否则返回-1。该函数的执行流程如图 21-5 所示。

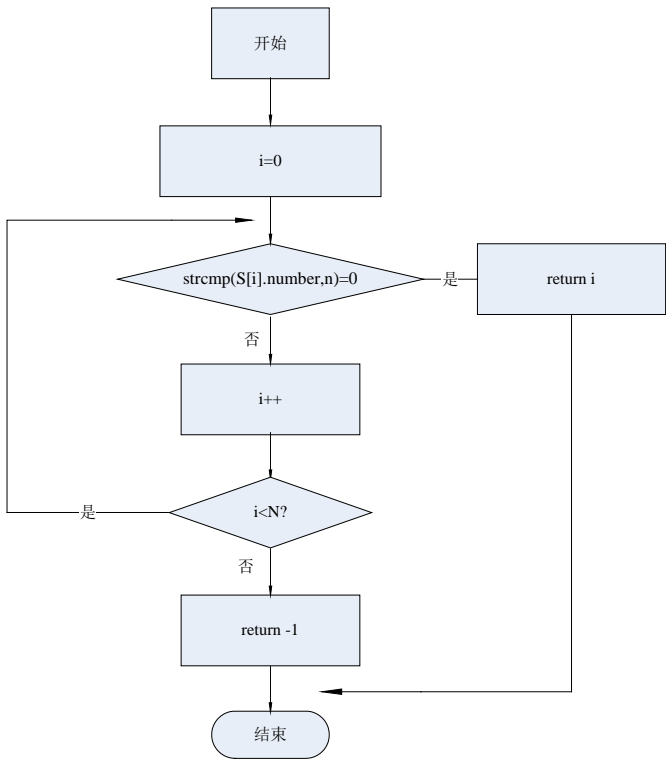


图 21-5 成绩查询流程图

 在上述第 5 行代码中，使用了字符串比较函数 strcmp()来实现两个学号的比较，因此在预处理中必须加上<string.h>头文件。

21.3.5 输出模块

事实上,如果对该系统内的成绩做了排序、统计和查询等功能后,都需要将结果输出。下面给出输出学生信息的函数 print()的实现代码,如代码清单 21-5 所示。

代码清单 21-5

```

1 void print(Student &s) //输出函数
2 {
3     cout<<"排名"<<"\t"<<"学号"<<"\t"<<"姓名"<<"\t"<<"语文:"<<"\t"
//定义输出格式
4     <<"数学:"<<"\t"<<"英语:"<<"\t"<<"总分"<<"\t"<<"平均分"<<endl;
5     cout<<s.rank<<"\t"<<s.number<<"\t"<<s.name<<"\t"<<s.chinese<<"\t"
//输出成员的值
6     <<s.math<<"\t"<<s.english<<"\t"<<s.total<<"\t"<<s.average<<endl;
7 }

```



读者可以看出,该函数中,输出了学生成绩管理系统中所有的成员值。在主函数 main()中调用成绩查询、统计等函数后都需要调用该函数,将结果输出到屏幕上。

21.4 系统集成

通过 21.3 节的介绍,将学生成绩管理系统的各个功能模块都实现了,接下来需要做的就是如何将模块集成起来,形成一个完整的系统。

根据前面的学习,读者知道,在 C++编写的程序中,首先执行的是主函数 main()。因此,系统的集成可以在 main()函数中实现。此外,读者知道,系统集成最好的方法是通过菜单的方式实现。因此,下面通过 C++实现模拟菜单的功能,实现代码如代码清单 21-6 所示。

代码清单 21-6

```

1 int main() //主函数
2 {
3     const int M=3; //定义常量并初始化
4     Student S[M]; //创建对象数组
5     sort(S,M); //调用排序函数(此处可注释掉)
6     int order=1; //定义变量并初始化
7     while(order!=4) //进入循环
8     { //画出命令菜单
9         cout<<"***** 命令菜单 *****"
*****<<endl;
10        cout<<"1.打印所有排名"<<endl;
11        cout<<"2.打印出成绩在全班平均分以上的学生名单和数据信息"<<endl;
12        cout<<"3.任意输入一个学号,查找出该学生在班级中的排名及其考试成绩"<<endl;
13        cout<<"4.退出系统"<<endl;
14        cout<<"*****"
*****<<endl;
15        cout<<"输入命令选择:";
16        cin>>order; //接收键盘输入
17        switch(order) //进入多分支选择结构
18        {
19            case 1: //输入命令选择 1
20            {
21                for(int j=0;j<M;j++) //输出全部学生的信息
22                    print(S[j]);
23            }

```




```

24         break;                                //跳出分支结构
25     case 2:                                    //输入命令选择 2
26     {
27         double compare=getAverage(S,M);        //求平均值
28         for(int k=0;k<M;k++)                    //在全部记录范围内查找
29             if(S[k].average>compare)           //输出成绩在全班平均分上的学生
30                 print(S[k]);                  //打印输出
31     }
32     break;                                    //跳出分支结构
33     case 3:                                    //输入命令选择 3
34     {
35         char code[Max];                        //定义字符数组
36         cout<<"输入您要查找的学号:";
37         cin>>code;                            //获取输入的学号
38         int result=search(S,M,code);           //调用查询函数
39         if(result==-1)                         //没有找到
40             cout<<"您输入的学号不存在!!!"<<endl;
41         else                                   //找到
42             print(S[result]);                  //输出该学生的信息及成绩
43     }
44     break;                                    //跳出分支结构
45     case 4:                                    //输入命令选择 3
46     break;                                    //退出程序
47     default:                                  //输入其他选择命令
48         cout<<"输入的命令不存在!!!"<<endl;    //返回错信息
49     }
50 }
51 return 0;
52 }

```

上述代码中，第 17~49 行使用了一个多分支选择语句 `switch...case` 语句，该语句用于根据用户输入不同选择执行不同的程序模块，用于实现不同的功能。此外，该程序段在第 7 行通过一个 `while` 循环，实现如果用户的选择不是退出，则可以允许用户进行无限次循环的选择。将上述代码单独运行，则返回系统的主界面，如图 21-6 所示。

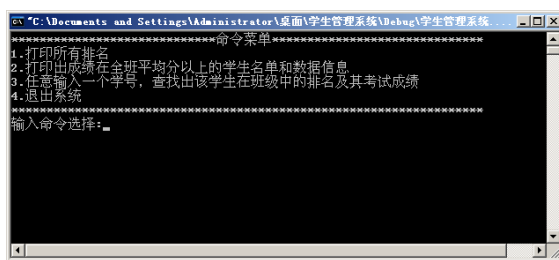


图 21-6 系统主界面

在上述代码中每个 `case` 语句后，跟的都是相对应的功能代码。例如，在 `case 1` 语句后调用 `print()` 函数打印出所有的排名，在 `case 2` 语句后调用 `getAverage()` 函数统计出平均成绩等。因此，在主函数 `main()` 中就实现了对 21.3 节所介绍的各个功能模块的集成。为了使读者更好地理解上述代码，下面给出其执行流程，如图 21-7 所示。



提示 读者从上述流程图可以看出，`main()` 函数可以根据用户选择的命令执行不同的程序模块，这就模拟了可视化程序中的菜单的功能。

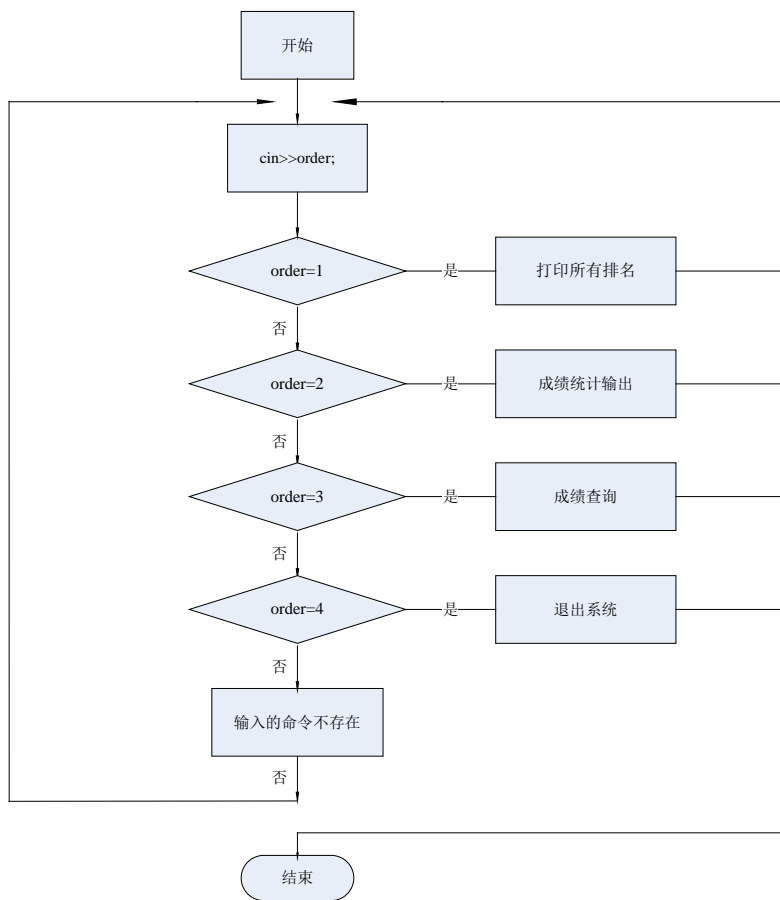


图 21-7 系统集成流程图

21.5 系统实现

经过上述功能模块实现和系统集成的介绍后，读者就可以将这些代码通过 Visual C++ 6.0 的编译器编译，并最终形成可执行程序。Visual C++ 6.0 中，可以通过控制台程序来实现该学生成绩管理系统。

Win32 控制台程序 (Win32 Console Application) 是一类 Windows 程序，它不使用复杂的图形用户界面，程序与用户交互时通过一个标准的正文窗口，通过几个标准的输入/输出流 (I/O Streams) 进行。本书所涉及的 C++ 源程序都可以在控制台程序下运行。

学生成绩管理系统通过控制台程序来实现，实现步骤如下所示。

① 新建 Win32 控制台程序。在 Visual C++ 6.0 继承开发环境中单击【File】/【New】命令，打开【New】对话框。在【Project】标签中，选中【Win32 Console Application】选项，输入工程名和路径，如图 21-8 所示。



注意 新建控制台程序后，建立的应用程序就能够直接双击运行，并给出运行结果。

② 选择控制台程序类型。在图 21-8 中单击【OK】按钮进入 Win32 Console Application-Step 1 of 1 对话框，在其中选择【A simple Application】选项后，单击【Finish】按钮完成向导，如



图 21-9 所示。

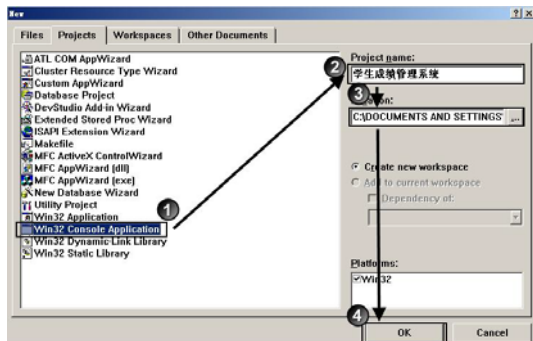


图 21-8 新建 Win32 控制台程序

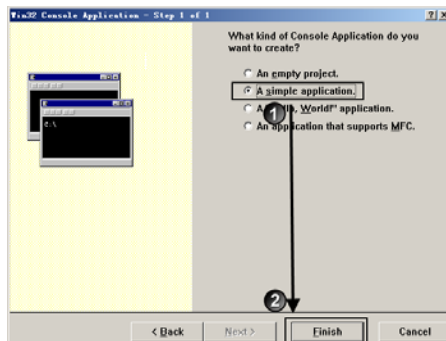


图 21-9 选择控制台程序类型

③ 完成向导。在图 21-9 中单击【Finish】按钮后，Visual C++ 将完成创建控制台程序的向导，并给出创建该工程的基本信息，如图 21-10 所示。

④ 输入代码。创建工程完成后，Visual C++ 6.0 的集成开发环境中只有工作区和输出窗口，在左侧的工作区中打开【ClassView】标签中的“学生成绩管理系统 classes”，打开其中包含的 main() 函数，右侧的代码编辑框中将显示该函数，如图 21-11 所示。



工程基本信息对话框给出的是用户通过 AppWizard 进行选择的结果和系统自动生成的结果，如图 21-11 中表示建立了“求素数.cpp”文件，头文件为 stdafx.h 和 stdafx.cpp。

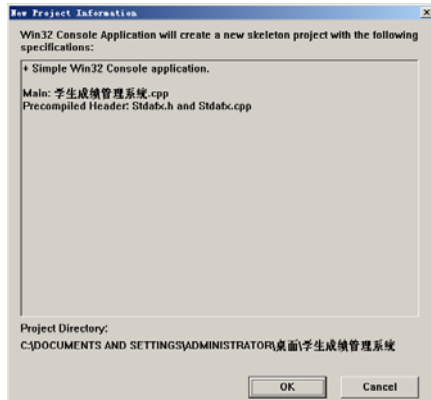


图 21-10 工程基本信息

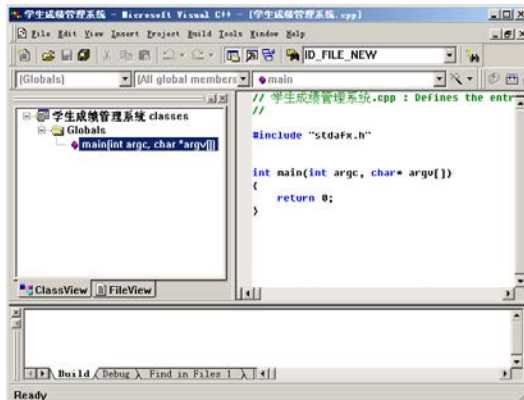


图 21-11 集成开发环境

在图 21-11 右侧的代码编辑框中，删除原来的 int main() 函数，输入整个学生成绩管理系统的实现代码。为方便读者阅读，此处将完成的代码分为几个部分。

21.5.1 结构和变量定义部分

该部分主要定义学生成绩管理系统中所用到的变量、常量和类，对于该系统将用到的函数，在该部分中给出函数声明。此外，该部分还包含了头文件的和命名空间，代码如代码清单 21-7 所示。

代码清单 21-7

```
1  #include<iostream>           //输入/输出库
2  #include<cstdio>             //对应C中的stdio.h头文件
3  using namespace std;         //使用命名空间
```



```
4   const int Max=30;                //定义字符串最大长度

5   class Student;                   //类声明
6   void setData(Student &s);        //设置对象 s 的数据
7   void count(Student &s);         //计算对象 s 的总分, 平均分
8   void sort(Student S[],int N);    //把长度为 N 的对象数组 S, 按平均分排序
9   double getAverage(Student S[],int N); //计算全班的平均分
10  void print(Student &s);          //打印信息
11  int search(Student S[],int N,char *n); //从长度为 M 的对象数组中, 查找学号 n 的位置
12  class Student                    //定义类
13  {
14  public:                          //定义公有成员
15      char number[Max];            //学号
16      char name[Max];              //姓名
17      double chinese;              //语文成绩
18      double math;                 //数学成绩
19      double english;              //英语成绩
20      double total;                //总成绩
21      double average;              //平均成绩
22      int rank;                    //排名
23  };
```

21.5.2 功能函数定义部分

该部分主要对上面部分中声明的函数进行具体的定义。在学生成绩管理系统中, 用到了录入成绩、统计成绩、计算平均成绩等函数, 这些函数功能的实现都在该部分中进行。各个函数的实现代码如代码清单 21-8 所示。

代码清单 21-8

```
1   void setData(Student &s)          //定义录入成绩函数
2   {
3       cout<<"输入学号,姓名,语文,数学,英语成绩:"; //录入数据
4       cin>>s.number>>s.name>>s.chinese>>s.math>>s.english;
5       s.total=0;                    //成员初始化
6       s.average=0;
7       s.rank=0;
8   }

9   void count(Student &s)            //定义统计成绩函数
10  {
11      s.total=s.chinese+s.math+s.english; //求一个学生总成绩
12      s.average=s.total/3;             //求一个学生平均成绩
13  }

14  void sort(Student S[],int N)        //插入法排序
15  {
16      int index;                      //定义变量
17      Student inserter;               //创建对象
18      for(int i=1;i<N;i++)            //依次比较所有学生
19      {
20          inserter=S[i];              //取出元素 i
21          index=i-1;
22          while(index>=0&&inserter.average>S[index].average) //该元素大于 i-1 个元素
23          {
24              S[index+1]=S[index];    //元素后移
```



```

25         index--; //继续比较
26     }
27     S[index+1]=inserter; //插入该元素 i
28 }
29 for(int j=0;j<N;j++)
30     S[j].rank=j+1; //设置排名
31 }

32 double getAverage(Student S[],int N) //定义计算全部平均成绩函数
33 {
34     double Average=0; //定义变量并初始化
35     for(int i=0;i<N;i++)
36         Average+=(S[i].chinese+S[i].math+S[i].english); //求全部成绩
37     Average/=(N*3); //求全部平均成绩
38     return Average; //返回该平均成绩
39 }

40 void print(Student &s) //定义打印输出函数
41 {
42     cout<<"排名"<<"\t"<<"学号"<<"\t"<<"姓名"<<"\t"<<"语文:"<<"\t"
43         //定义打印格式
44         <<"数学:"<<"\t"<<"英语:"<<"\t"<<"总分"<<"\t"<<"平均分"<<endl;
45     cout<<s.rank<<"\t"<<s.number<<"\t"<<s.name<<"\t"<<s.chinese<<"\t"
46         //定义输出成员值
47         <<s.math<<"\t"<<s.english<<"\t"<<s.total<<"\t"<<s.average<<endl;
48 }

49 int search(Student S[],int N,char *n) //定义成绩查询函数
50 {
51     for(int i=0;i<N;i++) //比较所有学生记录
52     {
53         if(strcmp(S[i].number,n)==0) //找到
54             return i; //返回该记录的位置
55     }
56     return -1; //没有找到
57 }

```

21.5.3 主函数部分

该部分是学生成绩管理系统的主界面部分。在该部分实现了对上述功能函数的调用，并给出了操作界面，使得用户可以与该系统进行交互，其实现代码如代码清单 21-9 所示。



提示 读者可以将代码 21-9 与上述代码 21-6 相比较，可以发现，下面代码运行后将先提示用户输入学生数据，而这正是该学生成绩管理系统运行其他操作的前提。

代码清单 21-9

```


1  int main() //主函数
2  {
3      const int M=3; //定义常量，即允许有 3 位同学
4      Student S[M]; //创建对象数组
5      for(int i=0;i<M;i++) //循环
6      {
7          cout<<"下面输入第"<<i+1<<"位同学的数据:"<<endl;
8          setData(S[i]); //输入所有 3 位同学的信息和成绩
9          count(S[i]); //计算每位同学平均的分和总分

```

```

10         cout<<endl;                                //输出换行
11     }
12     sort(S,M);                                       //排序
13     int order=1;                                     //定义变量并初始化
14     while(order!=4)                                  //order 为 4 时退出循环
15     {
16         cout<<"*****命令菜单*****"
*****<<endl;
17         cout<<"1.打印所有排名"<<endl;
18         cout<<"2.打印出成绩在全班平均分以上的学生名单和数据信息"<<endl;
19         cout<<"3.任意输入一个学号, 查找出该学生在班级中的排名及其考试成绩"<<endl;
20         cout<<"4.退出系统"<<endl;
21         cout<<"*****"
*****<<endl;
22         cout<<"输入命令选择:";
23         cin>>order;                                  //接收用户输入的命令选择
24         switch(order)                                //多分支语句
25         {
26             case 1:                                  //选择命令 1
27             {
28                 for(int j=0;j<M;j++)                //输出所有学生信息
29                     print(S[j]);
30             }
31             break;                                    //跳出分支语句
32             case 2:                                  //选择命令 2
33             {
34                 double compare=getAverage(S,M);      //获取全部平均值
35                 for(int k=0;k<M;k++)
36                     if(S[k].average>compare)         //平均成绩大于全部平均成绩
37                         print(S[k]);                 //输出
38             }
39             break;                                    //跳出分支语句
40             case 3:                                  //选择命令 3
41             {
42                 char code[Max];                      //定义字符数组, 用于存储学号
43                 cout<<"输入您要查找的学号:";
44                 cin>>code;                            //接收用户输入的学号
45                 int result=search(S,M,code);         //调用查询函数
46                 if(result==-1)                       //没有找到
47                     cout<<"您输入的学号不存在!!!"<<endl;
48                 else
49                     print(S[result]);                //输出该学生的信息及成绩
50             }
51             break;                                    //跳出分支语句
52             case 4:
53             break;                                    //跳出循环, 退出程序
54             default:
55                 cout<<"输入的命令不存在!!!"<<endl;  //返回错误
56             }
57         }
58         return 0;
59     }

```

编译运行。将上述代码输入完成后, 通过 Visual C++ 6.0 的快捷键【Ctrl+F7】可以对代码进行编译, 如有语法错误则不能通过编译。编译后, 通过快捷键【F7】建立应用程序, 再通过快捷键或工具栏中的  图标运行程序, 其运行结果如图 21-12 所示。

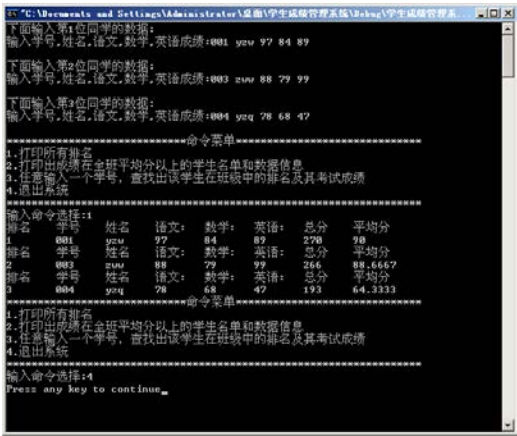


图 21-12 运行结果

21.6 小结

本章主要通过一个较为综合的范例——简单学生成绩管理系统的开发，来介绍了开发一个 C++ 应用程序的流程和一些技巧。首先，本章根据软件工程的生命周期，简单地介绍了系统的需求分析，然后讨论了系统的架构和需要的数据结构。接着，再以模块为单元依次介绍各个功能模块的实现，对于每个较为复杂的模块，都配以程序流程图，以便读者理解。最后，在 Visual C++ 6.0 环境下以控制台程序的方式实现该应用程序。本章的主要目的是让读者了解一个完整的 C++ 程序应该如何来建立，相信读者学习完后会有自己的理解。